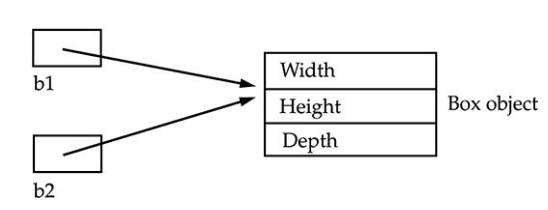
# Java memory allocation

* Declaring a variable just creates a reference
* new keyword creates an object in memory.
* Equals operator can change the reference to some other location

A diagram of a box

AI-generated content may be incorrect.

Whenever we use = operator with objects it does memory assignment.

* object1 = object2 it means both points to same memory location. When we change one object other will also get changed.
* If we do object1 = null object2 still points to previous memory location
* If we do object1 = object2 and object2 = object3 will mean object1 still points to old location and will not change, only object2 will point to object3
* If object1 = object2 and object1.property is changed then object2.property will also change

public class Test {

    public static void main(String[] args) {

        Box b1 = new Box(11, 12, 13);

        Box b2 = b1;

        b1.height = 24;     *// When one property of b1 is changed b2 also gets changed*

        System.out.println(b2.height);

        b1 = null; *// But when b1 is pointed to null b2 still points to previous object*

        System.out.println(b2.height);

    }

}

class Box {

    int height, width, length;

    public Box(int height, int width, int length) {

        this.height = height;

        this.width = width;

        this.length = length;

    }

}

# String

Strings are immutable, i.e. their values can’t be changed. When we assign new values to a string it just points to a new location in memory (string pool), the original string stays in the pool. When new operator is used new String is created then string pool is not used rather Heap memory is used.

String s1 = "Hello";

String s2 = "Hello";

System.out.println(s1 == s2);

s1 = new String("hello");

s2 = new String("hello");

System.out.println(s1 == s2);

What will be the output ?

String s1 = "Hello";

s1.concat(" World");

System.out.println(s1);

Since String is immutable in Java, whenever we do String manipulation like concatenation, substring, etc. it generates a new String and discards the older String for garbage collection. These are heavy operations and generate a lot of garbage in heap. So Java has provided StringBuffer and StringBuilder classes that should be used for String manipulation. StringBuffer and StringBuilder are mutable objects in Java

StringBuilder is not thread safe, StringBuffer is thread safe

For large no of concatenation use StringBuffer/StringBuilder

        int count = 100\_000;

        String s = "Hello";

        StringBuilder sb = new StringBuilder("Hello");

        StringBuffer sf = new StringBuffer("Hello");

        long  l1 = System.currentTimeMillis();

        for (int i = 0; i < count; i++) {

            s = s +i;

        }

        long l2 = System.currentTimeMillis();

        System.out.println("Time taken "+ (l2-l1));

        l1 = System.currentTimeMillis();

        for (int i = 0; i < count; i++) {

            sb = sb.append(i);

        }

        l2 = System.currentTimeMillis();

        System.out.println("Time taken "+ (l2-l1));

        l1 = System.currentTimeMillis();

        for (int i = 0; i < count; i++) {

            sf = sf.append(i);

        }

        l2 = System.currentTimeMillis();

        System.out.println("Time taken "+ (l2-l1));

# Object Class

Object class is by default extended by all other classes. Important methods exposed by Object class

**equals()** 🡪 This method compares the euqality of 2 objects. The default implementation uses == operator so it comared the memory location of 2 objects

**hashCode()** Objective of this method is to create a hash value from the object on which it is applied. The default implementation will creates a hash using the memory location of the object and will return integer

guidelines provided by Java in hashCode method documentation.

* If the equals method for 2 objects is returning true then hashcode of 2 objects should be same.
* If the hashCode for 2 objects are same it doesn’t gauranteee that the objects should be same
* For same object in same execution enviornment like same machine, OS etc. the hashcode should be same
* When we are overiding the hashCode() we should also also override the toString() method also and the parameters used for calculating hashCode, same should be used for toString also

**clone()** 🡪 In Java when we use “=” operator it doesn’t create clone of existing object, rather both the objects refer to same memory location, so when we change one variable other also gets changed. In order to create a clone of an object we have to use clone method provided in Object class which will create a new Object with same properties

This method is native and protected. In order to implement this the class has to impelemnt marker interface Clonable and in overiding method just call the super implementation

# Immutable class and Record

Immutable Objects are those whose state can’t be changed once it is created. Mutable is opposite of this. In Java immutable object are created by keeping the member variables as private final only keeping setter and not keeping any getters. We can mark the class as final also which will prevent any child class from extending it and set the values using constructors

Java 16 gave us an alternative for this by using record class which does the same thing with less boiler plate code

# Interface

Interface provides a way to define a contract that a class must follow to implement a certain set of behaviors. So, inside an Interface we can define all abstract methods without worrying about the actual implementation logic

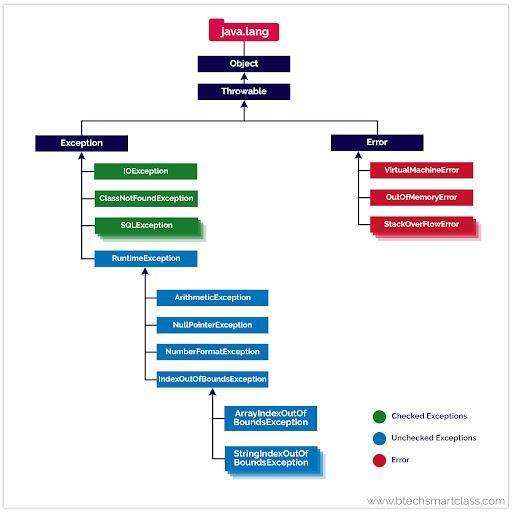
An interface can contain

1. Static final fields -> All field variables are public static and final by default even if we don’t mention it;
2. Abstract methods -> No Implementation abstract keyword is optional, can only be public
3. Default methods
4. Private methods
5. Static methods

Since abstract methods by default are public any Class overriding the method must be also public since it can’t have more restrictive access modifier

A **marker interface** is an interface in Java that does not contain any methods, fields, or constants. It is also known as a **tagging interface**. The primary purpose of a marker interface is to provide run-time type information about objects, so the compiler and JVM have additional information about the object. The **Serializable** and **Cloneable** interfaces are the example of marker interface. In short, it indicates a signal or command to the JVM. The declaration of marker interface is the same as interface in Java but the interface must be empty.

# Exceptions



# Collections

List 🡪 Array List When we want to access sequential data, maintains order of data, good for fetching data

Queue 🡪 Queue uses queue data structure like FIFO, LIFO example PriorityQueue, LinkedList (Also implements list)

Map 🡪 Key Value pair

Set 🡪 Unique Values

Weather to use LinkedList or ArrayList

If our use case involves fetching indexed data then ArrayList is better choice. If we need to insert data then Linked list will be better as insert will cause all element is List to be shifted

Java provides **Comparable** interface which should be implemented by any custom class if we want to use Arrays or Collections sorting methods. The Comparable interface has compareTo(T obj) method which is used by sorting methods. We should override this method in such a way that it returns a negative integer, zero, or a positive integer if “this” object is less than, equal to, or greater than the object passed as an argument.

Comparable.compareTo(Object o) method implementation can provide default sorting and we can’t change it dynamically. Whereas with **Comparator**, we can define multiple methods with different ways of sorting and then chose the sorting method based on our requirements

# JWT

In general, there are two types of tokens

1. **Opaque Tokens** are typically random strings with no inherent meaning. Used to reference authentication information stored on the server-side.
2. **JSON Web Tokens (JWT)** are self-contained tokens that consist of three parts: a header, a payload, and a signature, encoded in Base64 URL. Requires a call to the authorization server to validate and obtain user information.

|  |  |  |
| --- | --- | --- |
| Client will receive the token after successful login in a header/query string etc.  Client system must make sure to send the same token value on all the further request to the backend |  | Server will send token and send to client and stores client details and token in memory  Sever will validate the token and return result |

**Key Advantages of Token-Based Authentication**

**Security**

* Limited exposure of user credentials within the network
* Tokens can be revoked during suspicious activities without invalidating the user credentials

**Reusability**

* Tokens can be used across different domains and services, making them suitable for single sign-on (SSO) systems

**Cross-Platform Compatibility**

* Tokens can be used across various platforms and devices, including web applications, mobile apps, and IoT devices

**Expiration**

* Tokens can have specific expiration times set, ensuring tokens are valid only for a predefined duration

**Self-contained**

* Tokens are self-contained and carry all the necessary information about the user, roles/authorities etc.

**Statelessness**

* The token contains all the information to identify the user, eliminating the need for the session state
* If we use a load balancer, we can pass the user to any server, instead of being bound to the same server we logged in on
* JWT means JSON Web Token. It is a token implementation which will be in the JSON format and designed to use for the web requests.
* JWT tokens can be used both in the scenarios of Authorization/Authentication along with Information exchange which means you can share certain user related data in the token itself which will reduce the burden of maintaining such details in the sessions on the client/server side.

A JWT token has 3 parts each separated by a period(.) Below is a sample JWT token,

<https://jwt.io/>

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjQ3Mjg0MiIsIm5hbWUiOiJBbWl0IFNhaGEiLCJlbWFpbCI6ImFtaXQ4Ny5zQHRjcy5jb20ifQ.OWyFwQWMwXie7FNZa1mPp-BOZeRllUWwfigwmrllg8s

* + 1. Header
    2. Payload
    3. Signature (Optional)

**Header** Inside the JWT header, we store metadata/info related to the token. If we chose to sign the token, the header contains the name of the algorithm that generates the signature.

{

"alg": "HS256",

"typ": "JWT"

}

**Payload** In the body, we can store details related to user, roles etc. which can be used later for Authentication and Authorization. Though there is no such limitation what we can send and how much we can send in the body, but we should put our best efforts to keep it as light as possible.

{

"id": "472842",

"name": "Amit Saha",

"email": "amit87.s@tcs.com"

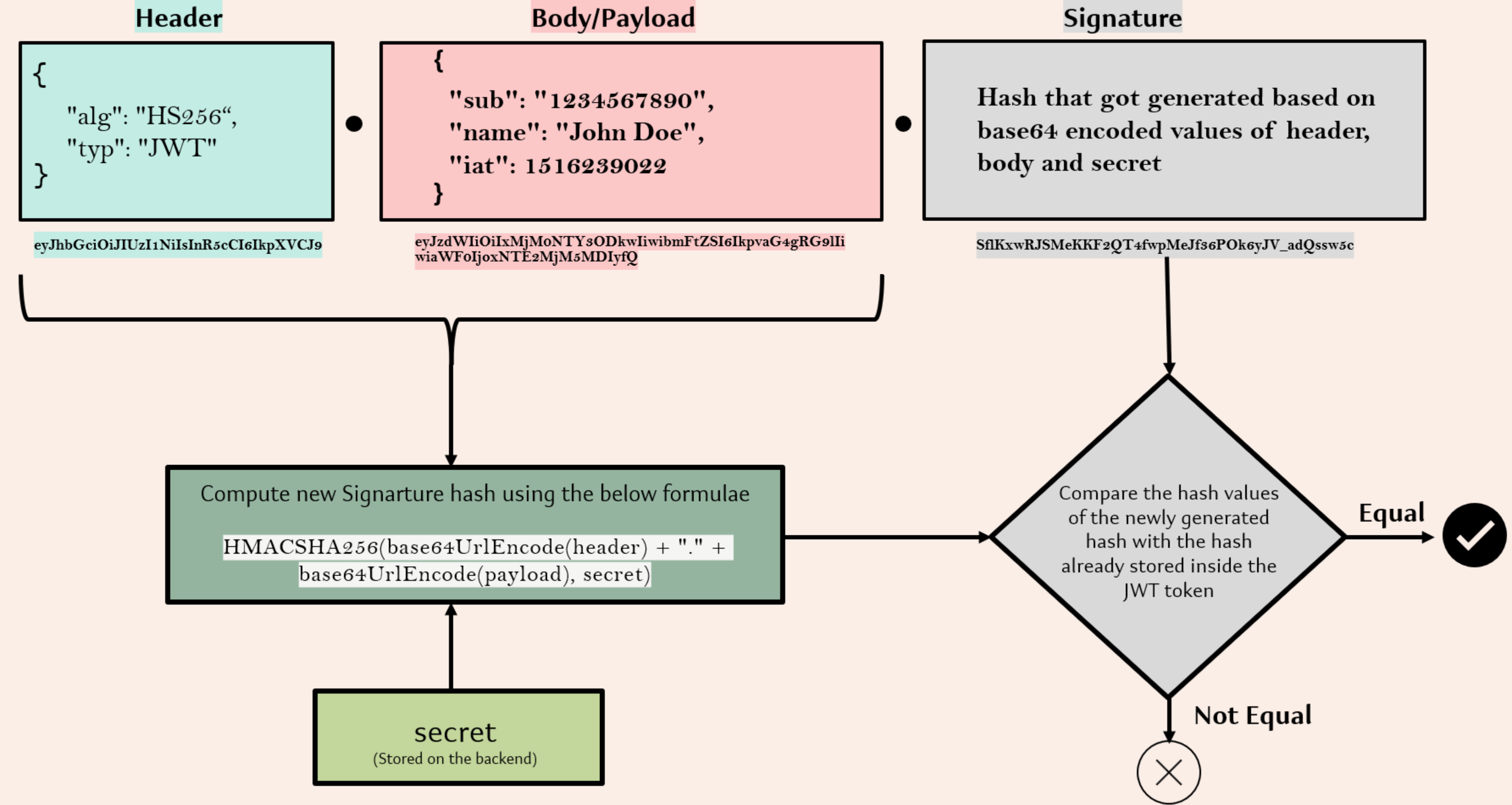
}

**Signature**

* To make sure that no one tampered the data on the network, we can send the signature of the content when initially the token is generated. To create the signature part, you must take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.
* For example, if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way

HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)

* The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.



# OAUTH2

[OAuth 2.0](https://tools.ietf.org/html/rfc6749), which stands for “Open Authorization”, is a standard designed to allow a website or application to access resources hosted by other web apps on behalf of a user.

OAuth 2.0 is an authorization protocol and NOT an authentication protocol

OAuth 2.0 uses Access Tokens. An **Access Token** is a piece of data that represents the authorization to access resources on behalf of the end-user. OAuth 2.0 doesn’t define a specific format for Access Tokens. However, in some contexts, the JSON Web Token (JWT) format is often used. This enables token issuers to include data in the token itself. Also, for security reasons, Access Tokens may have an expiration date.

**OAuth 2.0 Terminology - Standard Definitions**

**Resource Owner** The entity that can grant access to a protected resource. Typically, the end-user who owns the data (such as their email, profile, etc.). In OAuth flows, this is the person who authorizes an application to access their account.

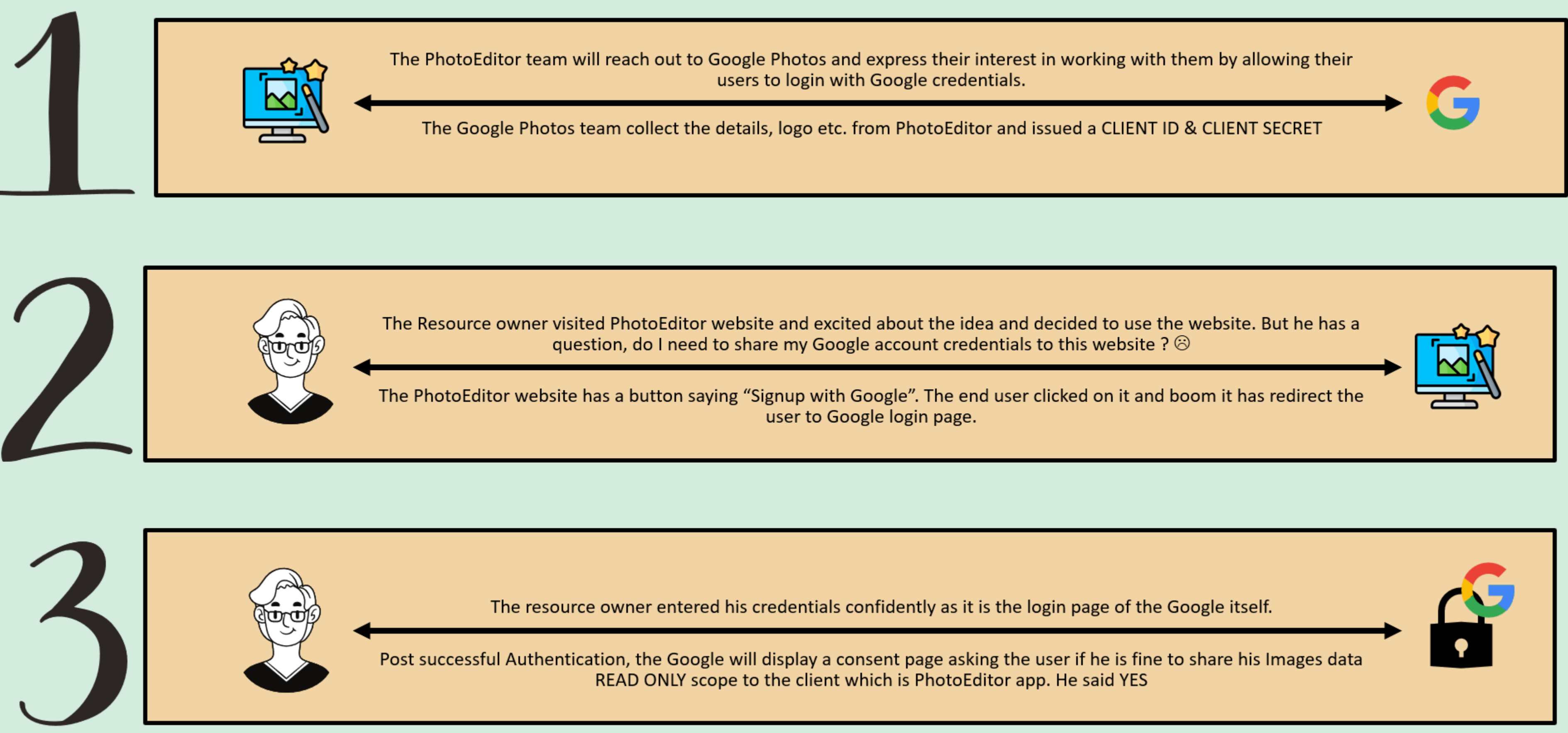
**Client** The application requesting access to protected resources on behalf of the Resource Owner. This can be a website, mobile app, desktop application, or any other software that needs to access protected resources but cannot directly use the Resource Owner's credentials.

**Authorization Server** The server that authenticates the Resource Owner, obtains authorization decisions, and issues access tokens. It's responsible for validating user identity and handling the OAuth authorization process. It maintains information about clients and their allowed access scopes.

**Resource Server** The server that hosts the protected resources the Client wants to access. It accepts and validates access tokens from the Client and serves the requested resources if the token grants appropriate permissions. Often, this may be the same physical server as the Authorization Server but with a distinct logical role.

**Scopes** Granular permissions that define the extent of access granted to the Client. Scopes limit what the Client can do with the access token. Examples include read-only access to emails, write access to calendar events, or access to profile information. The Authorization Server enforces these scope limitations when issuing tokens.

Sample OAUTH flow

A close-up of a text

AI-generated content may be incorrect.

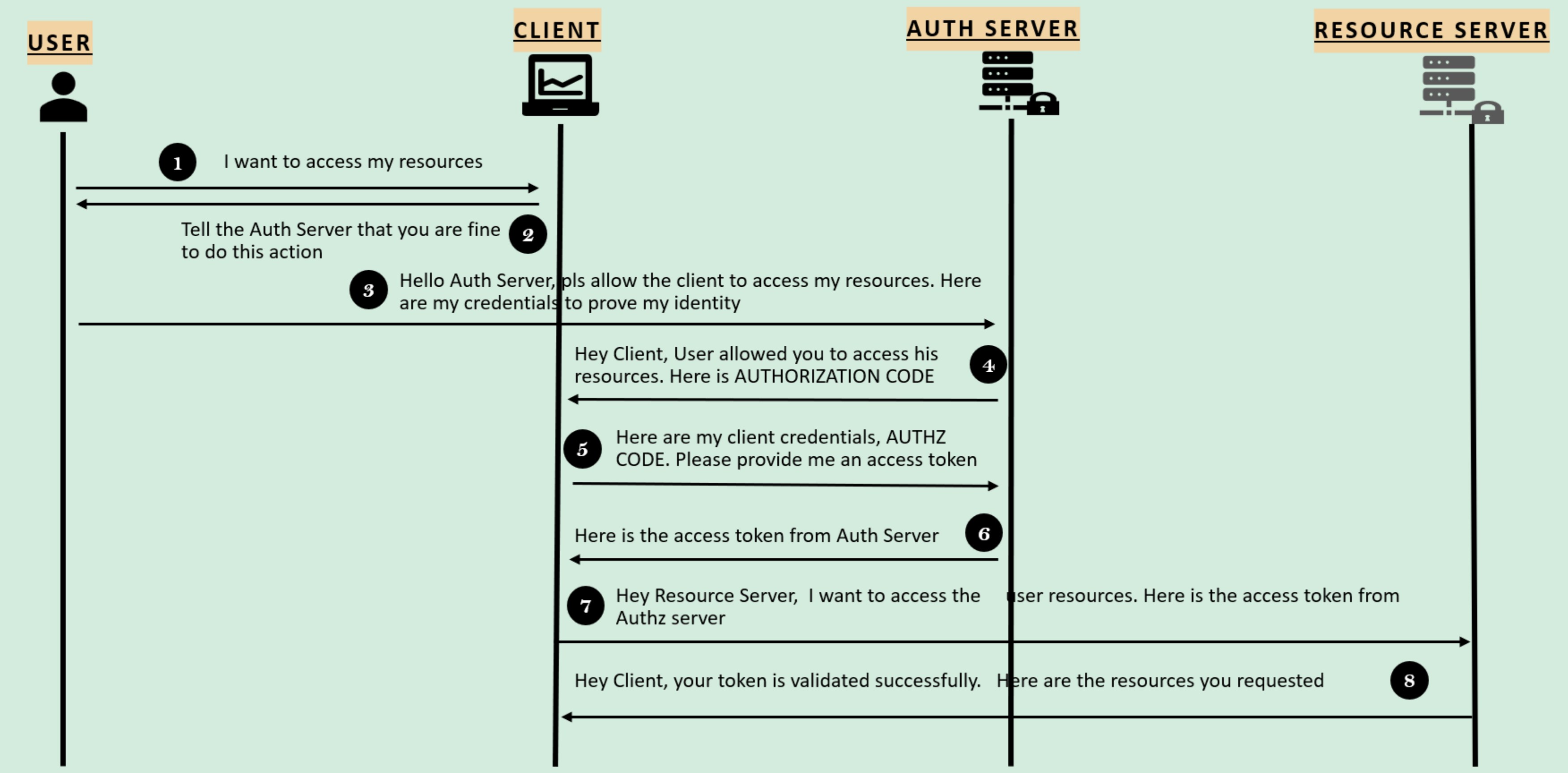
OAuth Grant Types

1. Authorization Code – When end user is involved
2. PKCE
3. Client Credentials
4. Device Code
5. Refresh Token
6. Legacy: Implicit Flow
7. Legacy: Password Grant

<https://www.oauth.com/playground/>

## Authorization Code

The Authorization Code grant is the most common OAuth 2.0 flow and is designed for applications that can securely store a client secret. It's optimized for security, particularly for web applications where the client can maintain confidentiality.



**Steps**

1. **Authorization Request**: The client redirects the user to the authorization server with its client ID, requested scope, a local state parameter, and a redirect URI.
2. **User Authentication & Consent**: The user logs in to the authorization server and approves the requested permissions.
3. **Authorization Code Response**: The authorization server redirects back to the client's redirect URI with a temporary authorization code and the original state parameter.
4. **Token Request**: The client sends a server-to-server request to the authorization server with the authorization code, client ID, client secret, and redirect URI.
5. **Token Response**: The authorization server validates the request and returns an access token, typically with a refresh token.
6. **Resource Access**: The client uses the access token to request protected resources from the resource server.

In the step 3, where client is making a request to Auth Server endpoint, must send the below important details,

* **client\_id** — the id which identifies the client application by the Auth Server. This will be granted when the client registers first time with the Auth server.
* **redirect\_uri** — the URI value which the Auth server needs to redirect post successful authentication. If a default value is provided during the registration, then this value is optional
* **scope** - like authorities. Specifies level of access that client is requesting like READ
* **state** — CSRF token value to protect from CSRF attacks
* **response\_type** — With the value ‘token’ which indicates that we want to follow implicit grant type

In the step 5 where client after received an authorization code from Auth server, it will again make a request to Auth server for a token with the below values,

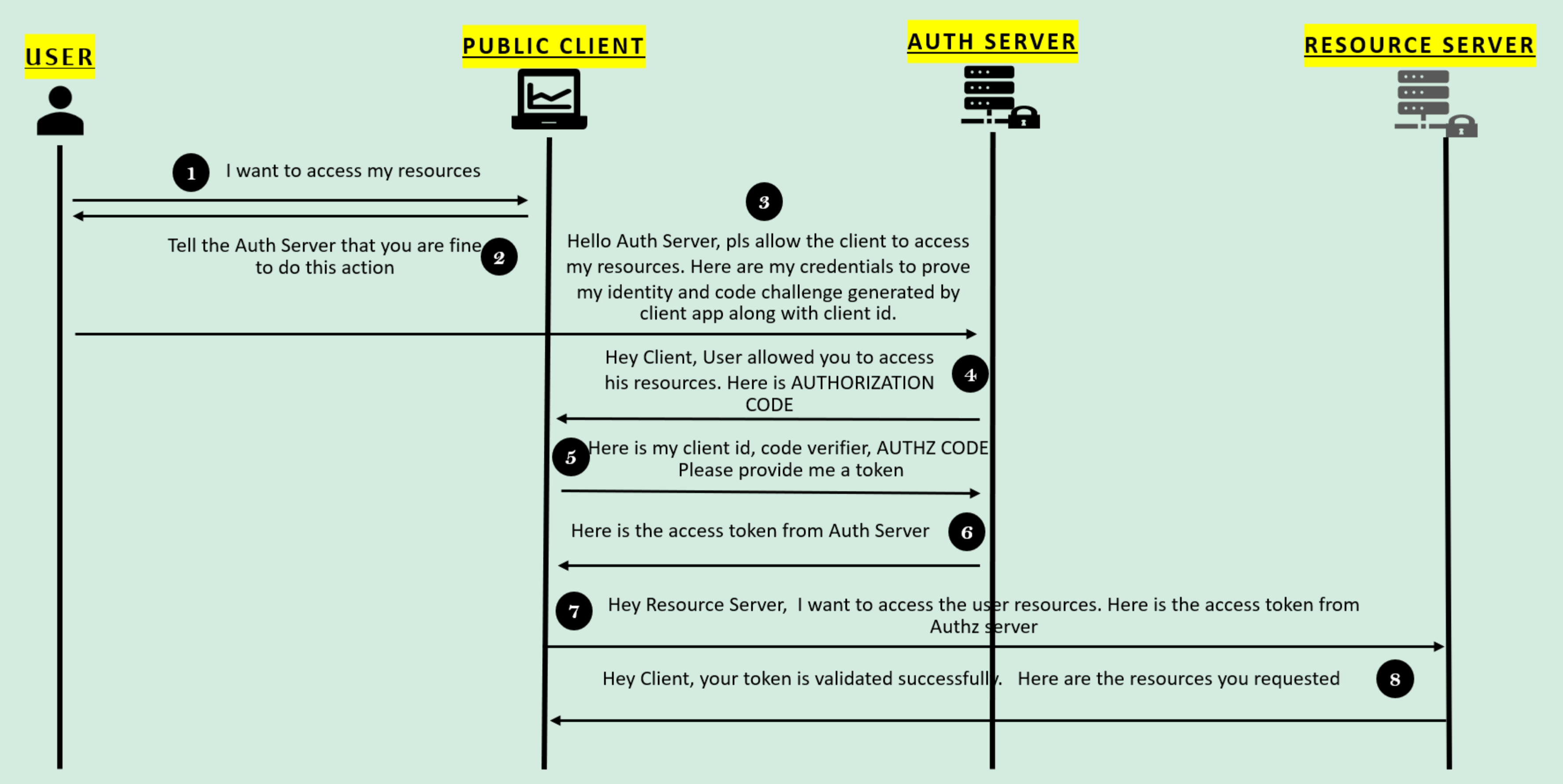
* code - the authorization code received from the above steps
* client\_id & client\_secret — the client credentials which are registered with the auth server. Please note that these are not user credentials
* grant\_type — With the value ‘authorization\_code’ which identifies the kind of grant type is used
* redirect\_uri

**Security Features**

* Authorization code is short-lived and single-use
* Full authentication flow happens server-to-server, not in the browser
* State parameter protects against CSRF attacks
* PKCE extension available for additional security with public clients
* Separates authentication from token issuance

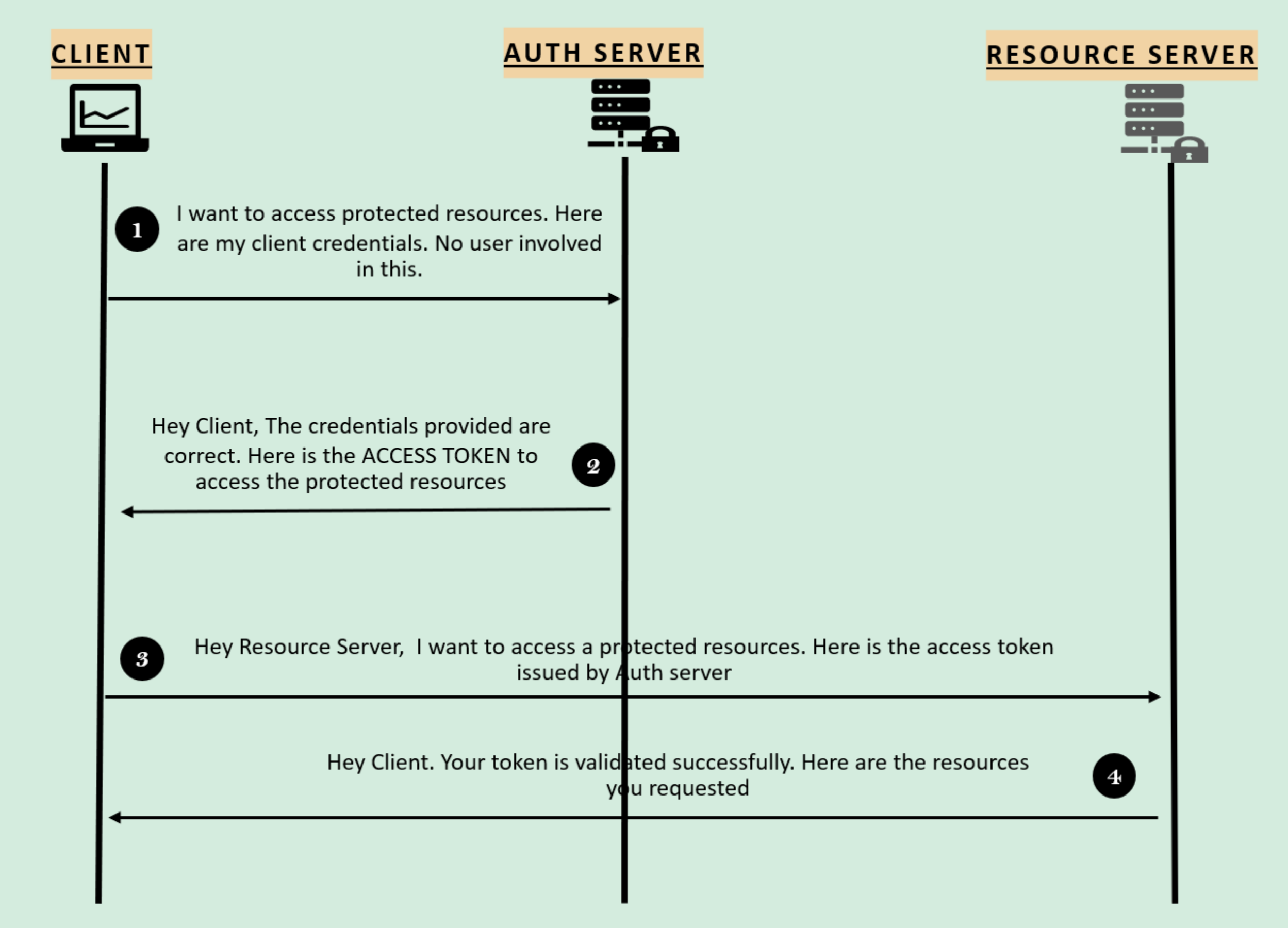
## PKCE

* When public clients (e.g., native and single-page applications) request Access Tokens, some additional security concerns are posed that are not mitigated by the Authorization Code Flow alone. This is because public clients cannot securely store a Client Secret.
* Given these situations, OAuth 2.0 provides a version of the Authorization Code Flow for public client applications which makes use of a Proof Key for Code Exchange (PKCE).
* The PKCE-enhanced Authorization Code Flow follows below steps,
  + Once user clicks login, client app creates a cryptographically random code\_verifier and from this generates a code\_challenge.
  + code challenge is a Base64-URL-encoded string of the SHA256 hash of the code verifier.
  + Redirects the user to the Authorization Server along with the code\_challenge.
  + Authorization Server stores the code\_challenge and redirects the user back to the application with an authorization code, which is good for one use.
  + Client App sends the authorization code and the code\_verifier(created in step 1) to the Authorization Server.
  + Authorization Server verifies the code\_challenge and code\_verifier. If they are valid it responds with ID Token and Access Token (and optionally, a Refresh Token).



## Client Credentials

The Client Credentials grant flow is one of the OAuth 2.0 authorization flows specifically designed for server-to-server authentication when there's no user context involved. It's the simplest OAuth 2.0 grant type and is used when the client itself is the resource owner.



**Steps**

1. The client (usually a server/service) authenticates with the authorization server using its client ID and client secret.
2. The authorization server validates these credentials.
3. If valid, the authorization server returns an access token to the client.
4. The client uses this access token to access protected resources on the resource server.

**Security Considerations**

* The client secret must be kept secure and never exposed to public clients
* Usually limited to confidential clients (servers that can securely store credentials)
* Scopes should be strictly limited to only what's necessary
* No refresh tokens are typically issued since the client can always request a new token

This flow is ideal for backend services that need to communicate with other APIs without user interaction, providing a secure authentication mechanism for service-to-service communication.

# Spring Boot

Why do we need DI and IOC container or why do we need Spring to create Beans?

Without Spring the beans would have to be created by application code with new operator. This creates 2 problems

1. Tight coupeling
2. Incorrect use of Heap memory. With new all the classes like DAO, Controller for each requests are created in Heap. Which is unnecessary as they don’t hold data

How Spring solves the issue. We don’t create the objects. The objects provides its dependedncy with Annotations and Spring container takes care of creating the beans and injecting dependency . So the task of creating beans and dependency is shifted to Framewor rather than application (Inverion of control)

How do we create beans

1. If it is a applation class created by us we can mark it as @Component or @Controller, @Service etc for specific types of classes
2. If we are using external library we can’t control the source code then we make @Configuration class and there we can have @Bean Method to return the instance

Bean Scope

* singleton
* prototype
* request
* session
* application

The last three scopes request, session, application , are only available in a web-aware application.

* Singleton: only one instance of the bean is created and shared across the entire application. This is the default scope.
* Prototype: a new instance of the bean is created every time it is requested

Key Features of Spring Boot

* Auto-Configuration
* Starter Dependencies
* **Embedded Servers**
* **Externalized Configuration**
* **Spring Boot Actuator**