Western Norway
University of
Applied Sciences

DAT152 – Advanced Web Applications

Authentication and Authorization

Part2 – Restful APIs

Goal for Today

Understand what is protected resource

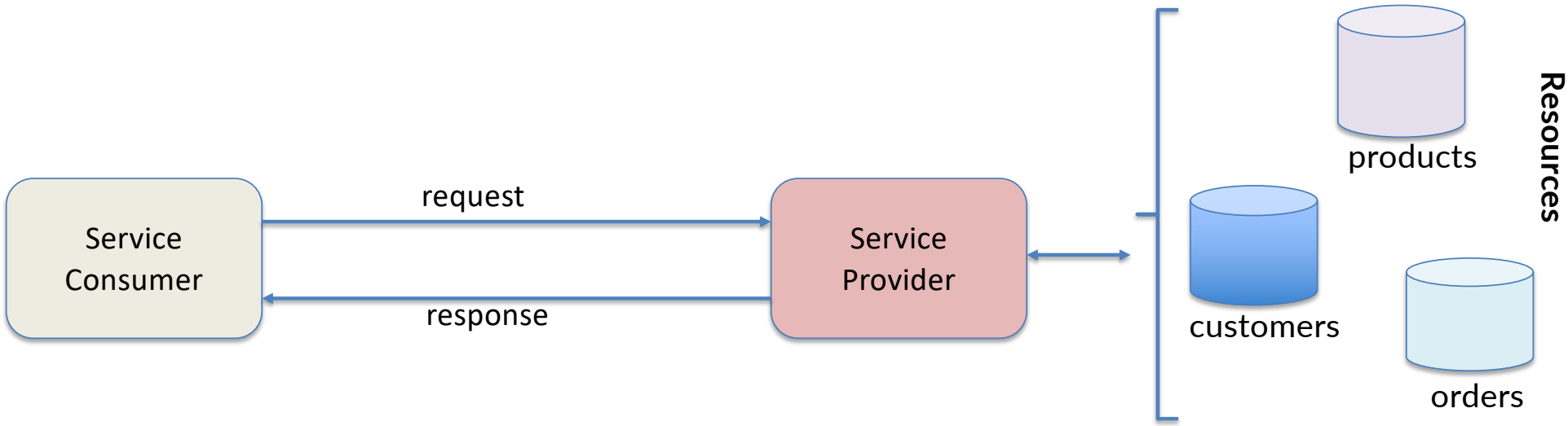
Understand how to protect REST API resources

Understand the Oauth2 authorization framework

Understand implementations of the schemes

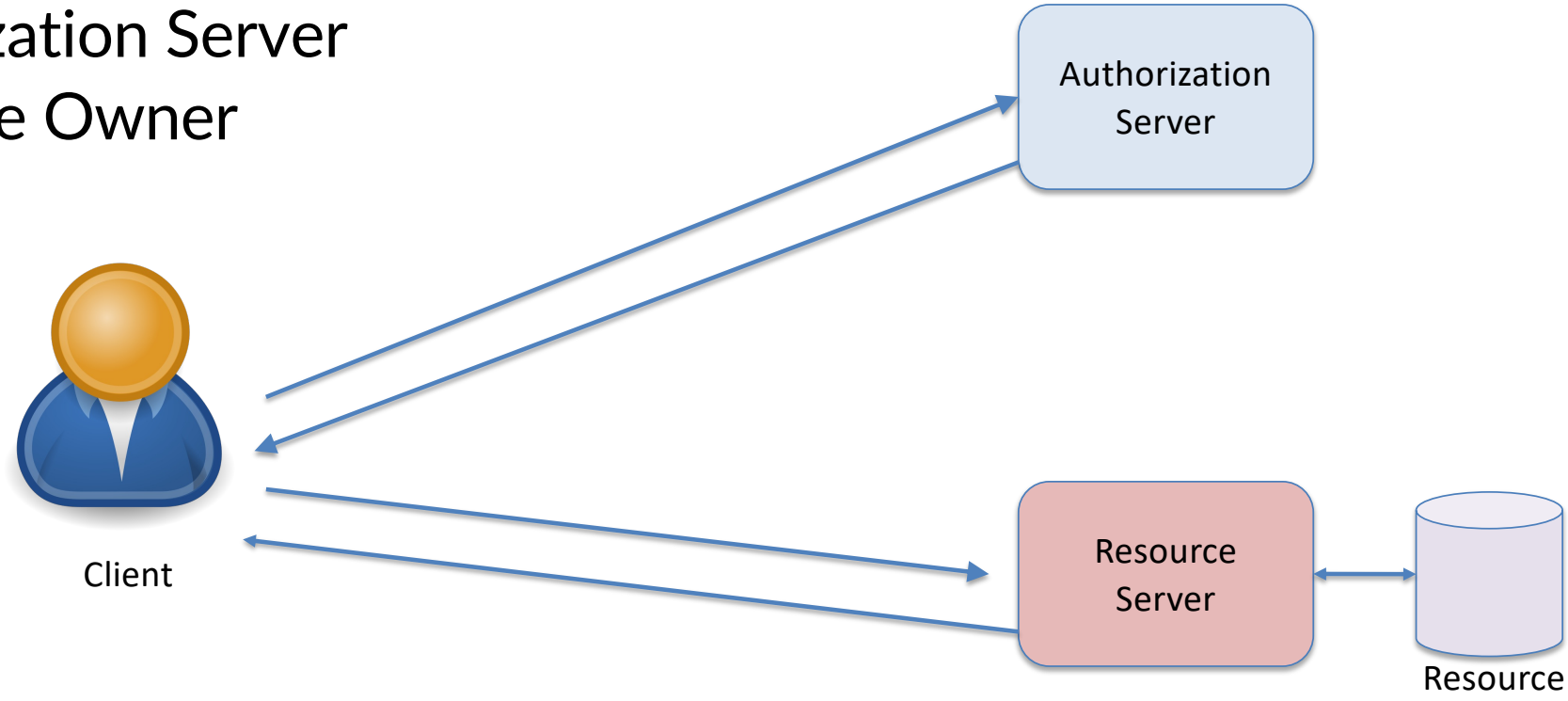
Protected Resources

- Authentication
  - Proof of identity – Verifies the identity of a user
- Authorization
  - Level of privilege – What are the access rights for this user?



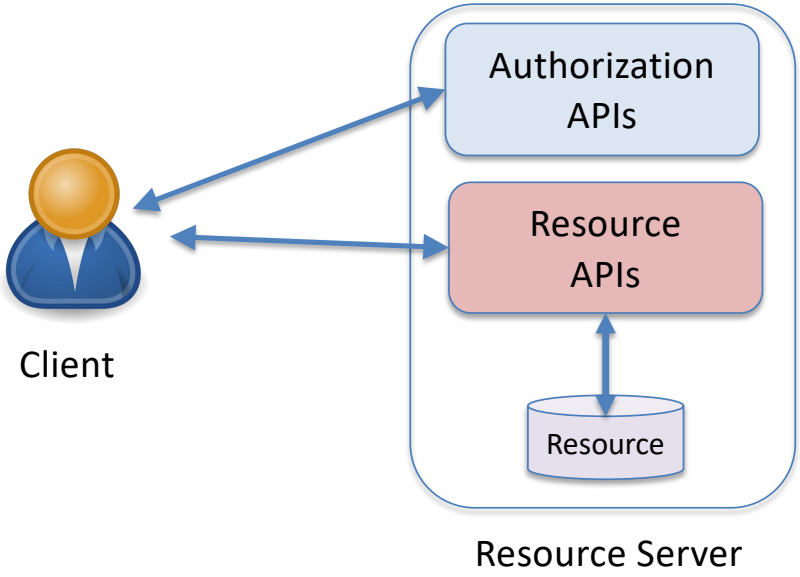
Parties

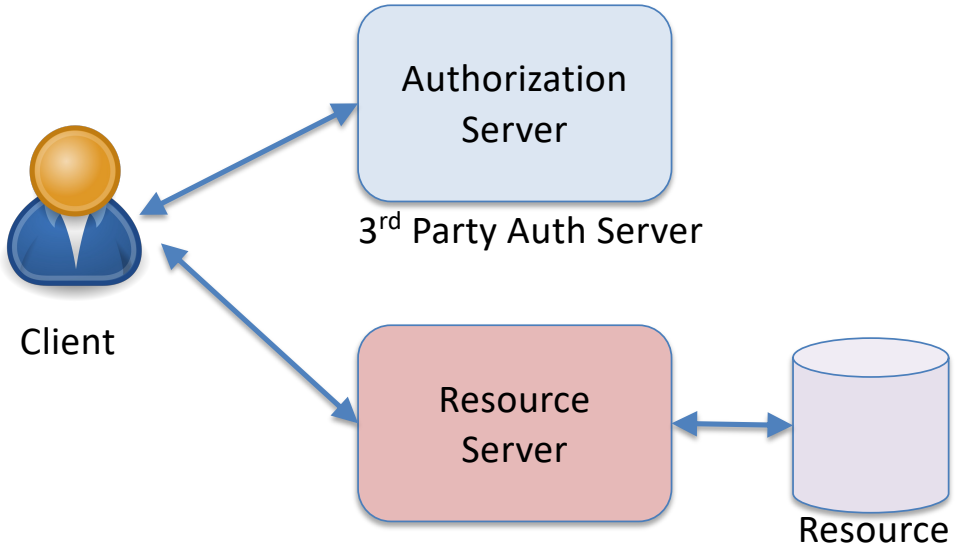
- Resource Server
- Authorization Server
- Resource Owner
- Client



Parties and Architectures

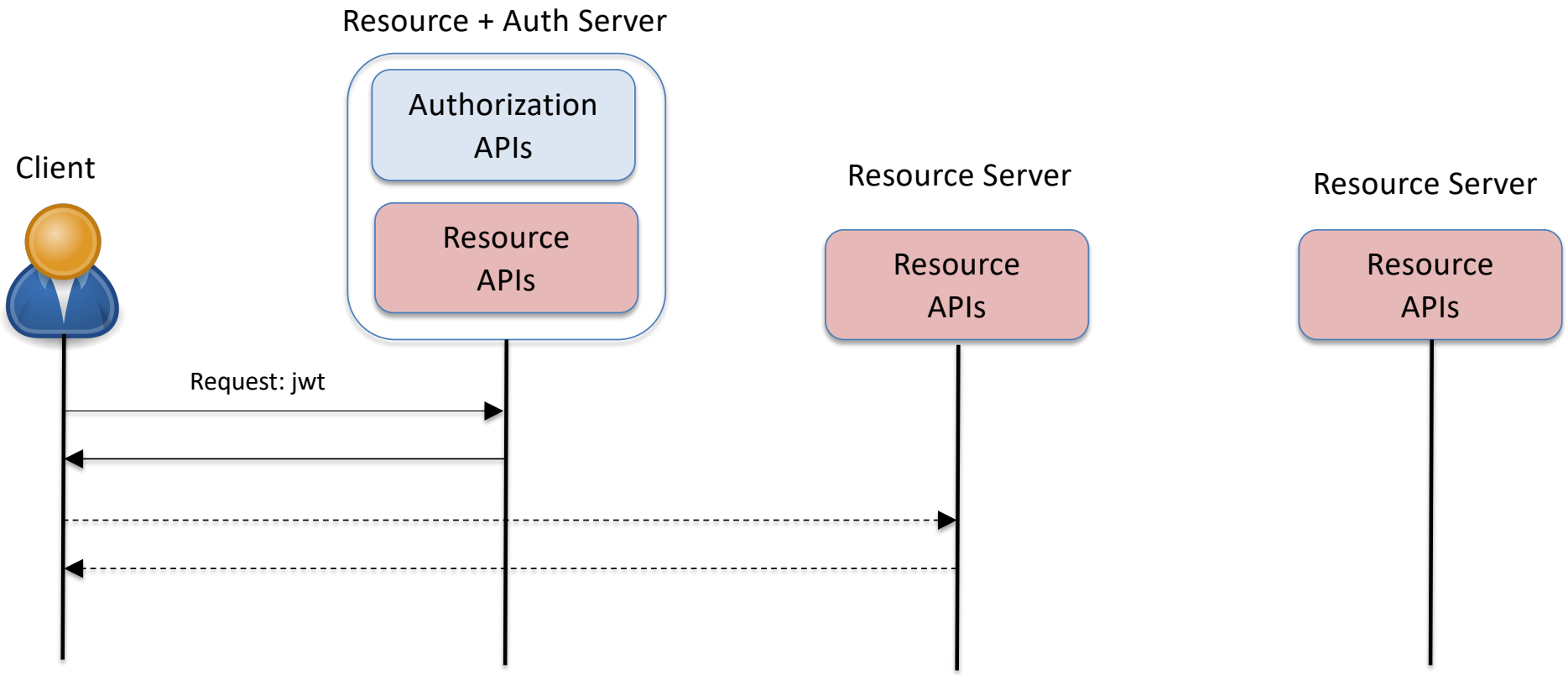
- A resource server can provide 'own' auth/authz functionality
- A resource server can delegate auth/authz to a 3rd Party identity management system





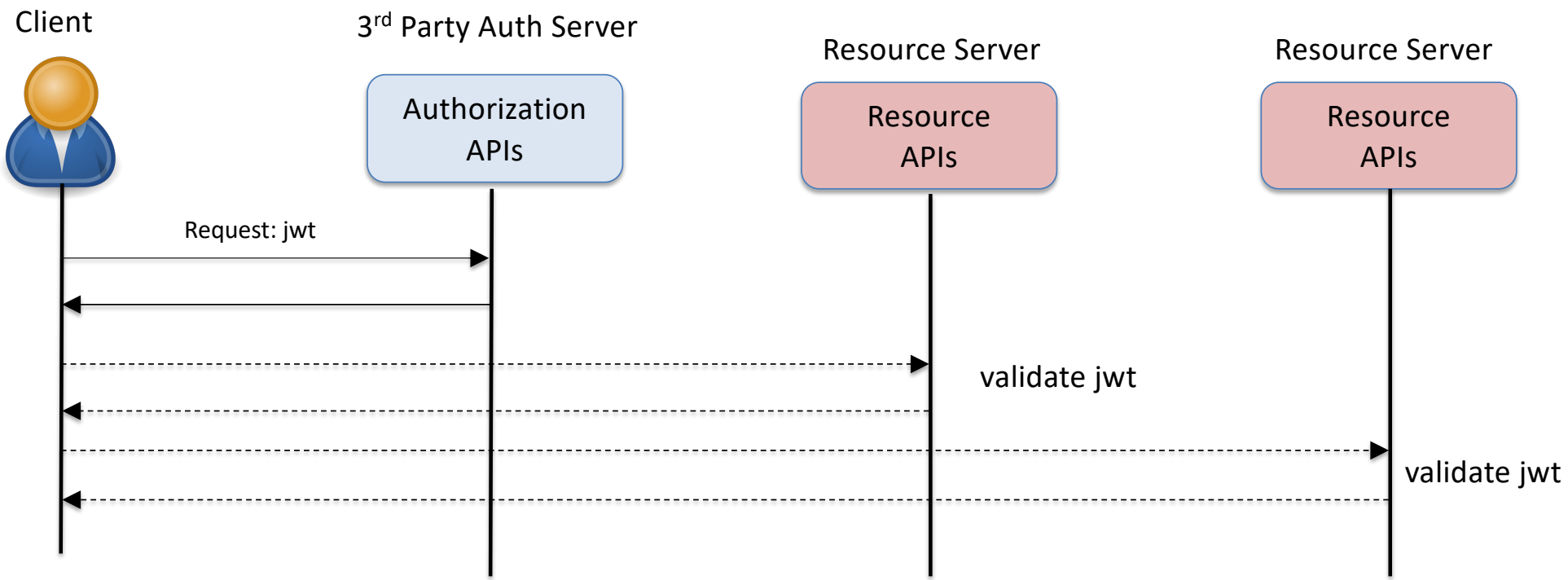
Parties and Architectures

- Resource servers can be replicated for high availability and performance



Parties and Architectures

- Resource servers can be replicated for high availability and performance



Stateless RESTful Web Services

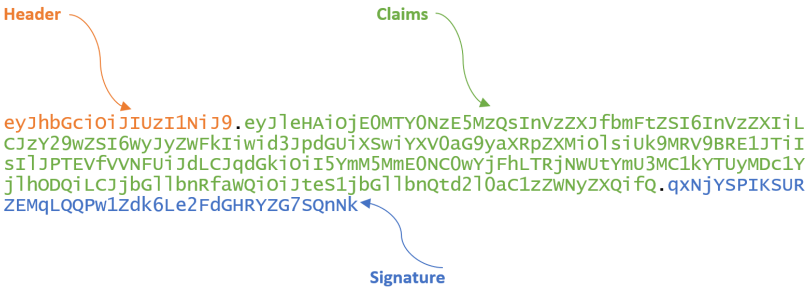
- This constraint dictates that each request from client to server must contain all of the information necessary for the server to understand the request
  - The server will not store anything about the latest HTTP request the client made
  - It will treat every request as new
  - No session, no history on the server
  - Session state is kept entirely on the client

Question

- What does this stateless constraint really mean with regards to the auth/authz server architectures?

Json Web Token (JWT)

Jwt-Token = Header.Claims.Signature



HEADER:

```
{
 "alg": "HS256»
}
```

PAYLOAD:

```
{
"iat": 1572038943,
"admin": "true",
"user": "Jerry»
}
```

SIGNATURE

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

Trust between parties

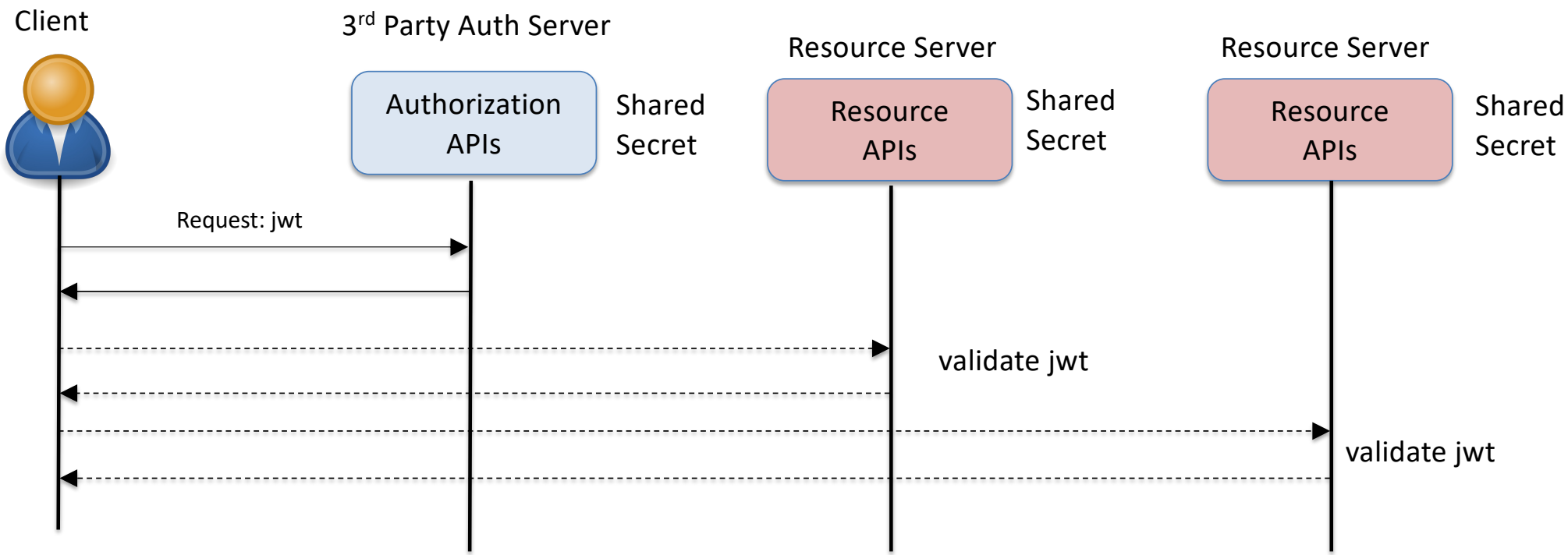
- JWT allows 2 parties to share claims based on trust
- Trust is established based on cryptography scheme
- There are two ways to establish trust between systems
  1. Use shared-key (secret), known as symmetric encryption
  2. Use public key, known as asymmetric encryption

Questions

- What are the challenges you can identify in both schemes?

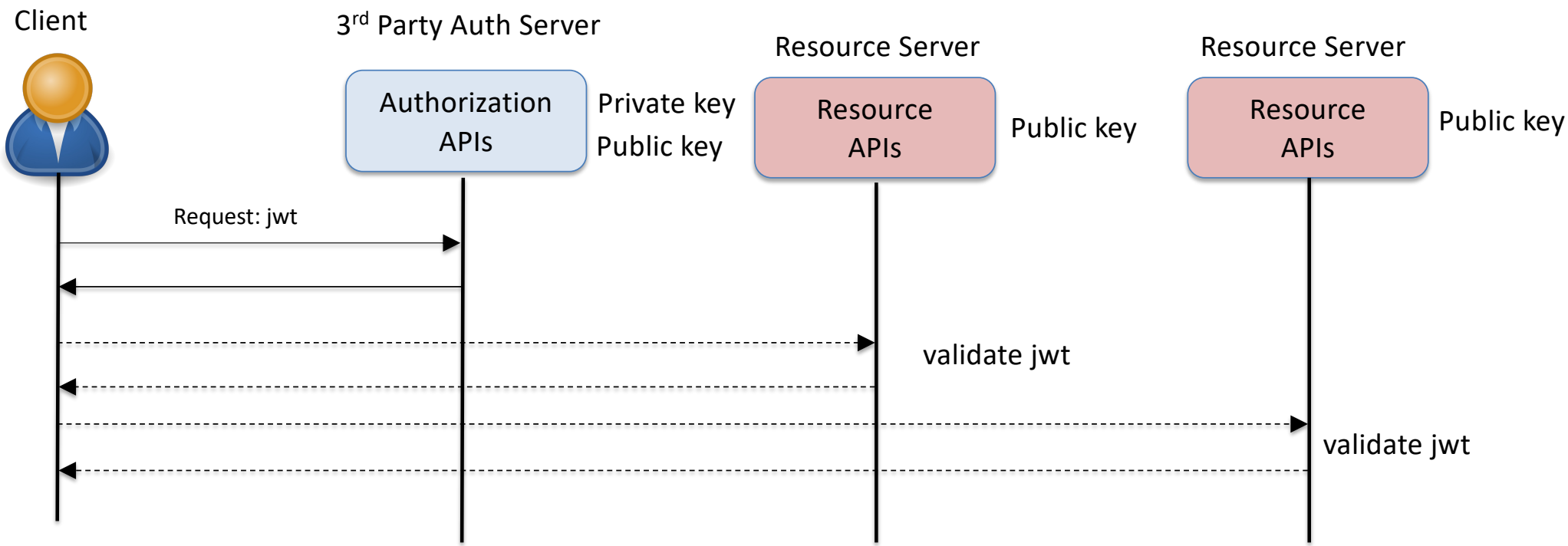
Authentication – Shared Secret

- Resource servers can be replicated for high availability and performance



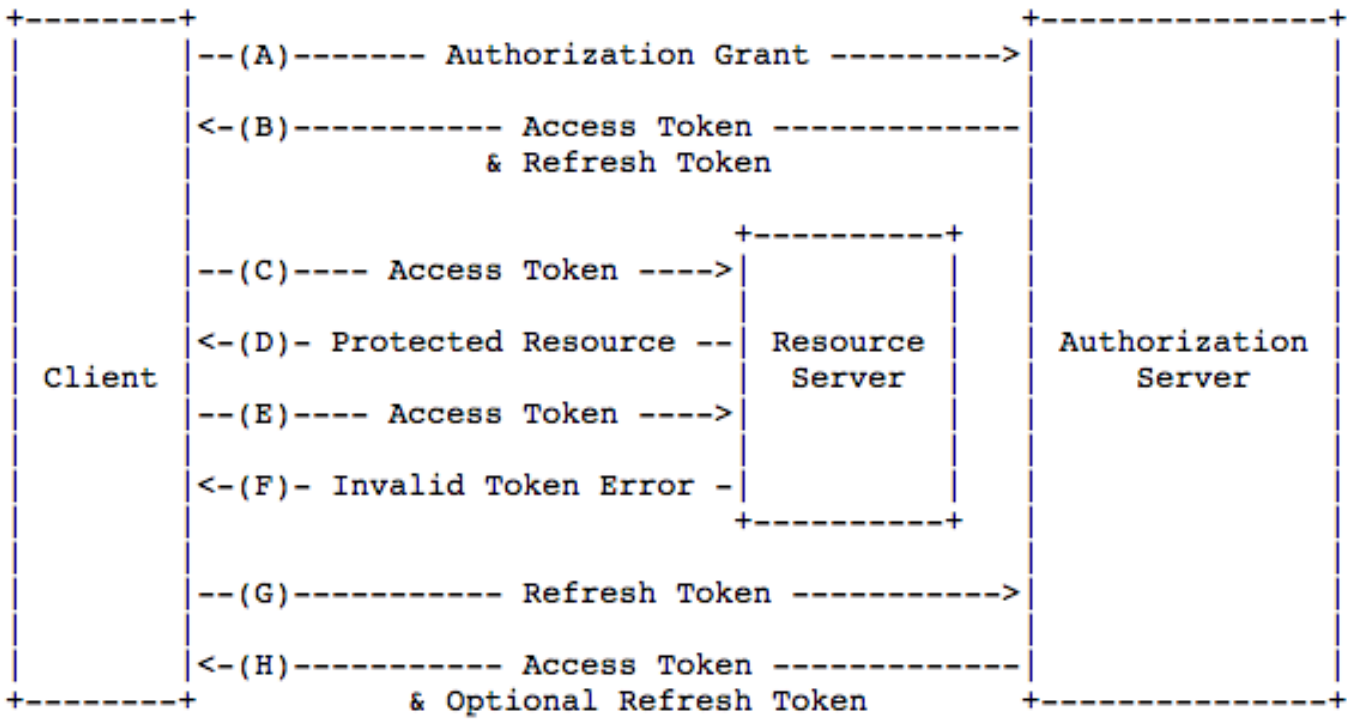
Authentication – Public Key

- Resource servers can be replicated for high availability and performance



OAuth2 Framework

- Oauth framework allows a client to negotiate an access token from an authorization server in order to access a protected resource
- Heavily depends on Json Web Token



OAuth2 Framework

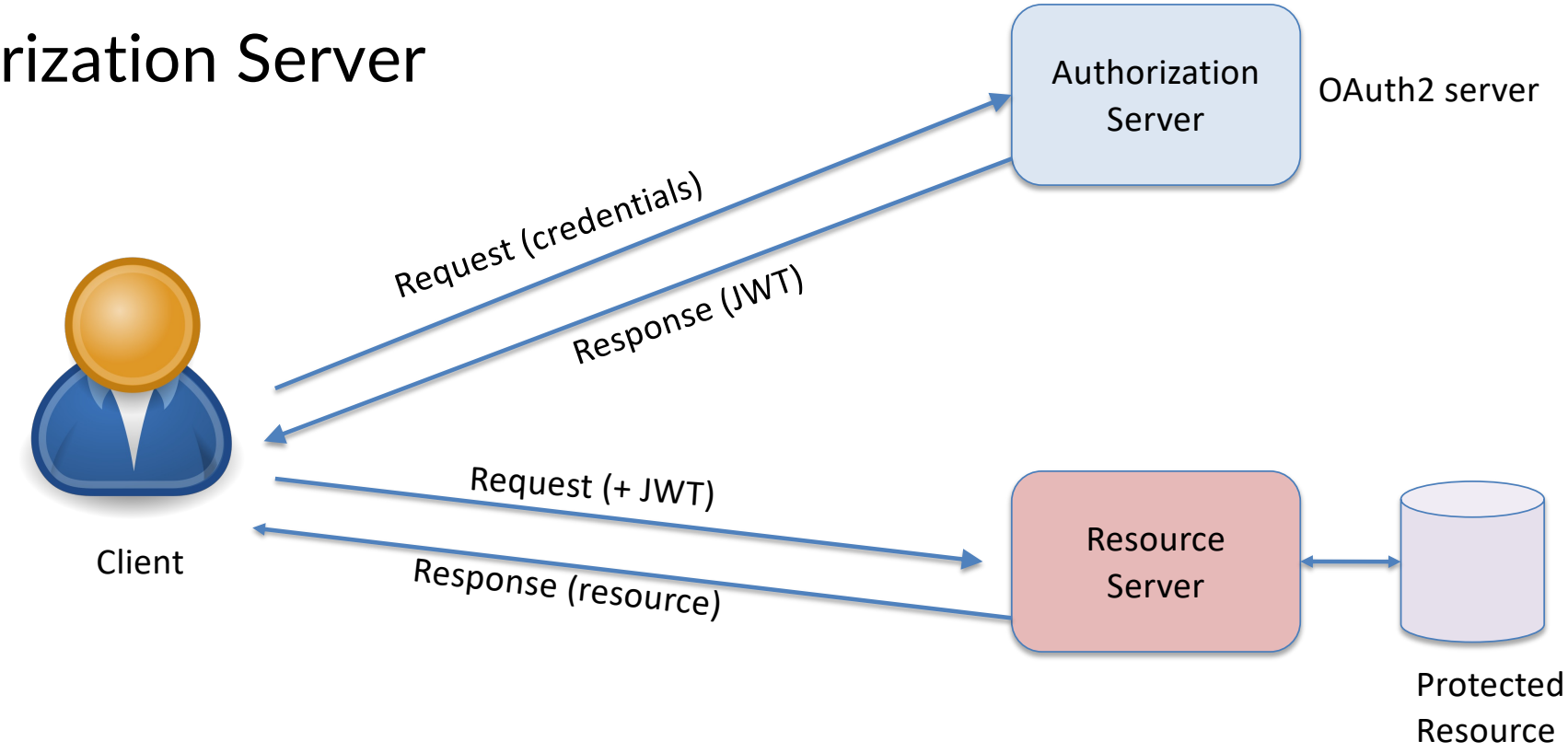
- In OAuth 2.0, **grants** are the set of steps a Client has to perform to get resource access authorization.
- Grant Types
  - Authorization code Flow
  - Implicit Flow
  - Resource Owner Credentials Grant
  - Client Credentials Grant Type
  - Device Authorization Flow
  - Refresh Token Grant

https://www.rfc-editor.org/rfc/rfc6750.html#page-5

https://auth0.com/intro-to-iam/what-is-oauth-2

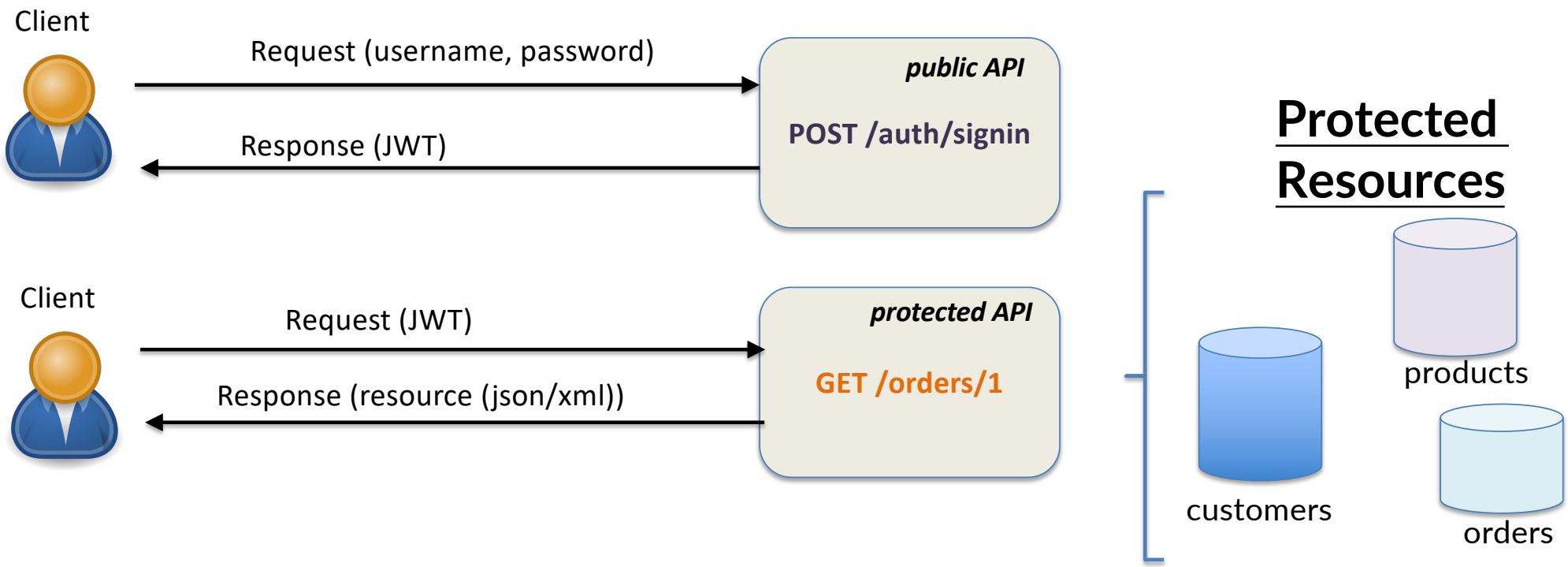
OAuth2 Server

- Resource Server
- Authorization Server
- Client



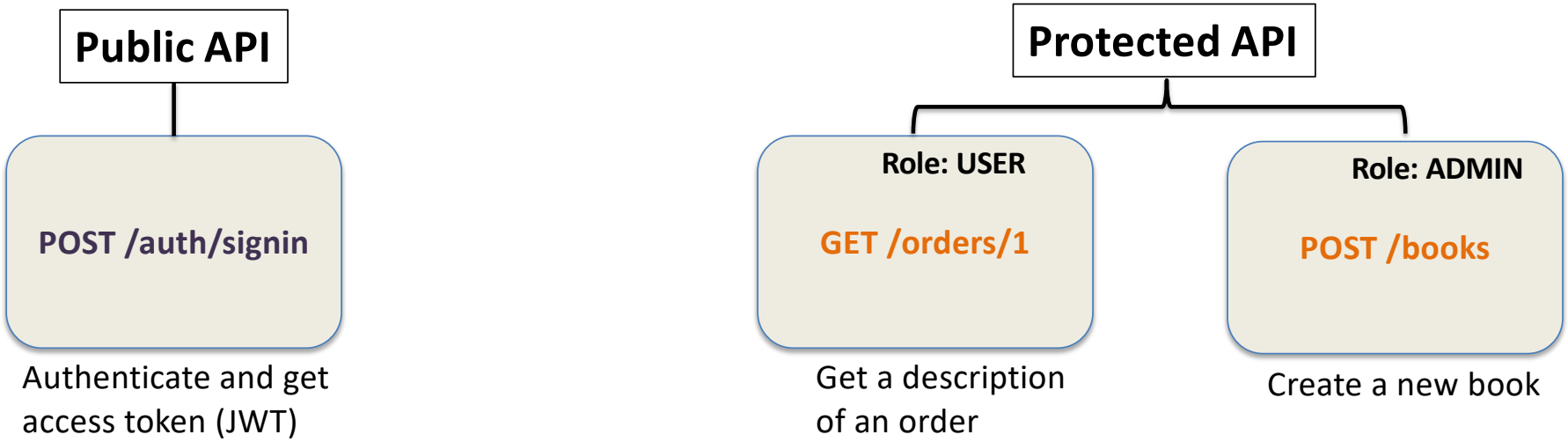
Protecting REST API Resources

- To protect resources, we can protect the specific REST API endpoints that interact with the resources



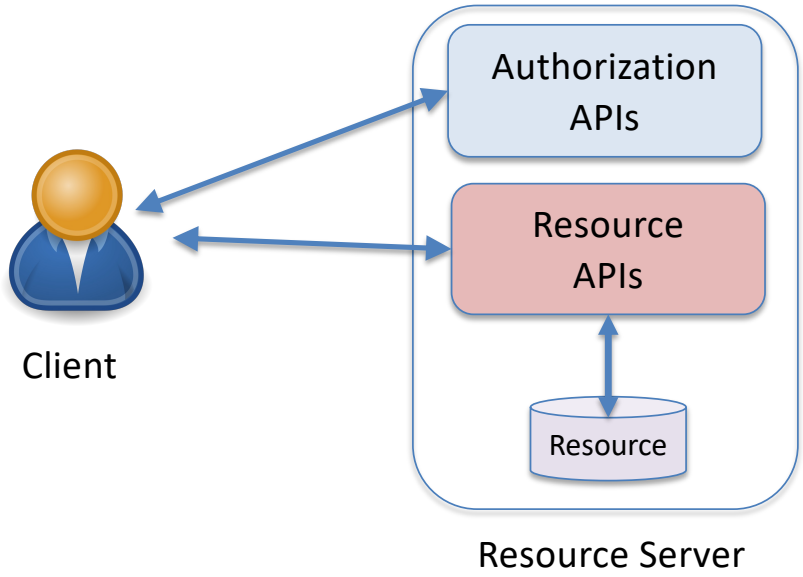
Authorization

- Authorization is about access roles
  - The level of privilege that a client has to different resources
- Privilege concerns what
  - type of resources a client has access to
  - actions a client can perform on them
- We'll look at how to implement Role-based Authorization



Authorization + Resource Server Design

- Auth server maintains and manages client details
  - Credentials (e.g. username, password, etc)
  - Access roles (SUPER, ADMIN, USER)




```
@Entity
public class User {
    ...
    private String email;
    private String password;
    ...
    @ManyToMany
    @JoinTable(name = "user_roles",
    joinColumns = @JoinColumn(name = "user_id"),
    inverseJoinColumns = @JoinColumn(name =
    "role_id"))
    private Set<Role> roles = new HashSet<>();
    ...

}
```

```
@Entity
@Table(name = "roles")
public class Role {
    private String name;
    ...
}
```

Authorization + Resource Server Design

- Authz server creates an access token (JWT) with all necessary claims

```
public String createAccessToken(Authentication authentication) {
    UserDetailsImpl userPrincipal = (UserDetailsImpl) authentication.getPrincipal();
    return Jwts.builder()
            .setSubject(userPrincipal.getEmail())
            .setIssuer("DAT152-Lecturer@TDOY")
            .claim("firstname", userPrincipal.getFirstname())
            .claim("lastname", userPrincipal.getLastname())
            .claim("roles", userPrincipal.getAuthorities().stream()
                    .map(role -> role.getAuthority())
                    .collect(Collectors.toList()))
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + EXPIRE_DURATION))
            .signWith(key(), SignatureAlgorithm.HS256)
            .compact();
}
```

Authorization + Resource Server Design

- Auth server sends JWT to client
- When client sends back JWT with request
- Auth server performs validation and build userDetails before proceeding to the action

```
public boolean validateAccessToken(String token) {
    try {
        Jwts.parserBuilder().setSigningKey(SECRET_KEY).build().parse(token);
        return true;
    }catch(MalformedJwtException ex) {
        LOGGER.error("JWT is invalid!", ex.getMessage());
    }catch(ExpiredJwtException ex) {
        LOGGER.error("JWT token is expired!", ex.getMessage());
    }
    ...
}
```

Auth/Authz Flow

```
curl -X POST http://localhost:8090/elibrary/api/v1/auth/signin -d '{"email":”user1@email.com","password":”user1"}'
```

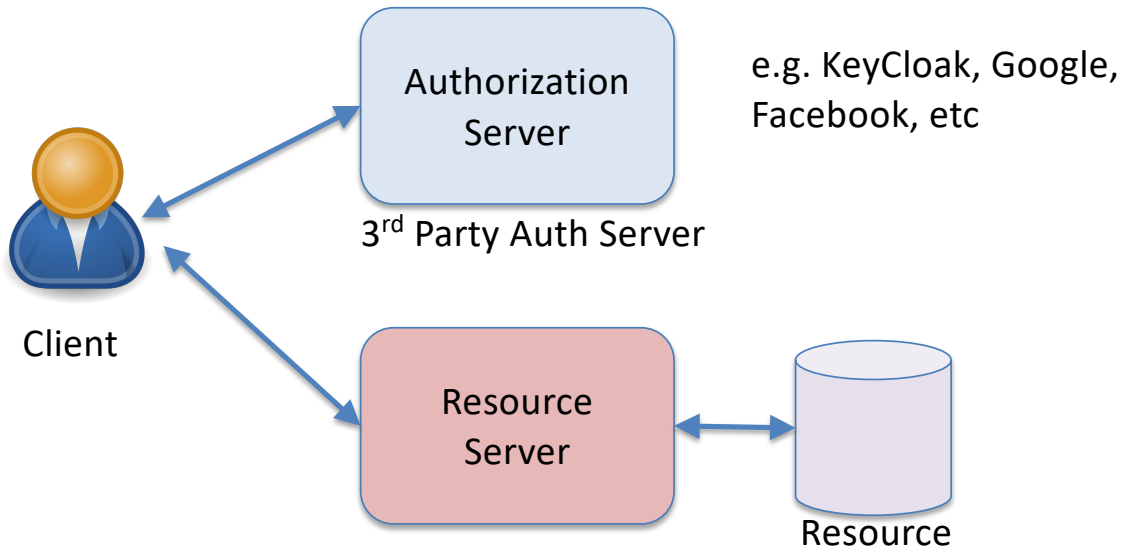

```
{
    "email": ”user1@email.com",
    "accessToken":
        "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJyb2JlcnRAZW1haWwuY29tIiwiaXNzIjoiREFUMTUyLUxlY3R1cmVyQFRET1kiLC
        JmaXJzdG5hbWUiOiJSb2JlcnQiLCJsYXN0bmFtZSI6IklzYWFjIiwicm9sZXMiOlsiVVNFUiJdLCJpYXQiOjE2OTYwNDEy
        ODEsImV4cCI6MTY5NjEyNzY4MX0.NDoPBqNiQNlxIF8mmnjxMJ_QQfrmVPU6H38Ez1fsg-c",
    "type": "Bearer"
}
```


```
curl -v -H "Authorization: Bearer <accessToken>" http://localhost:8090/elibrary/api/v1/orders/2
```


```
{
    "id": 2,
    "isbn": "abcde1234",
    "expiry": "2023-10-21"
}
```

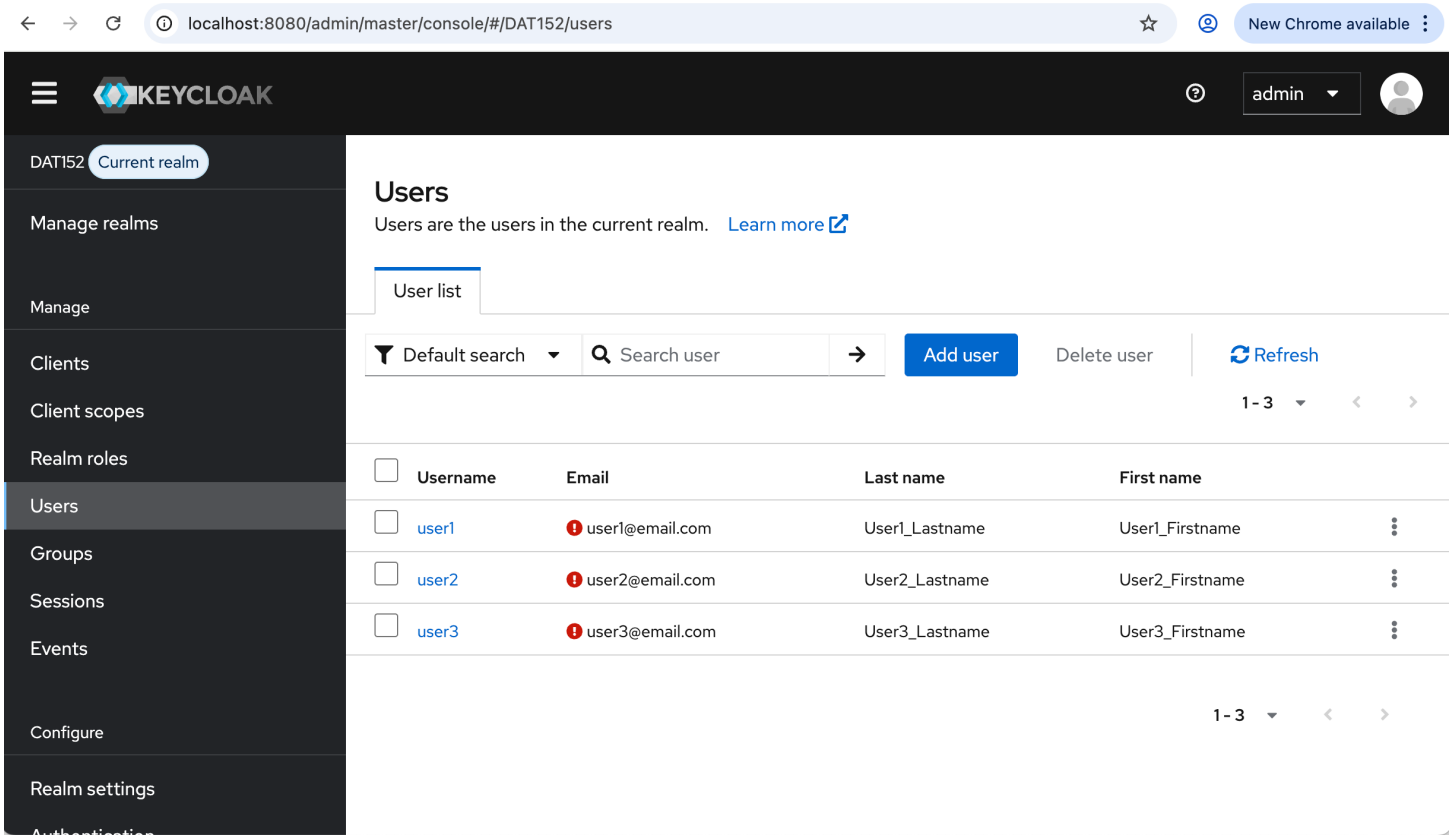
3rd Party Authorization Server

- In this scheme, a 3rd party IdP maintains and manages client's credentials
  - Credentials (e.g. username, password, etc)
  - Access roles



3rd Party Authorization Server

- Example: Keycloak IdP



3rd Party Authorization Server

- Resource server must have access to the public key (certificate) from the IdP (KeyCloak)
- Configure the resource server to use the public key to verify JWT

```
spring.security.oauth2.resourceserver.jwt.jwk-set-
uri=http://localhost:8080/realms/DAT152/protocol/openid-connect/certs
```

Auth/Authz Flow

```
curl -X POST http://localhost:8080/realms/DAT152/protocol/openid-connect/token --data
'grant_type=password&client_id=dat152oblig2&username=user1&password=user1'
```

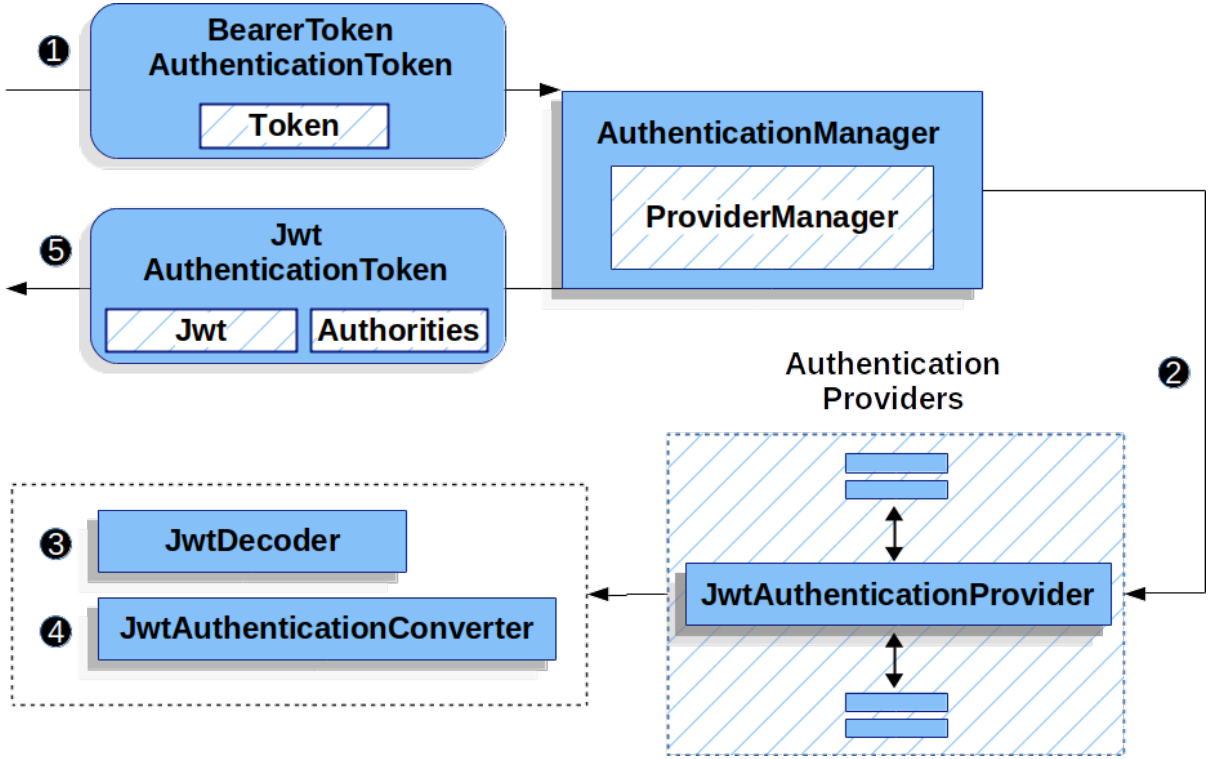

```
{
    "access_token":
        "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJyb2JlcnRAZW1haWwuY29tIiwiaXNzIjoiREFUMTUyLUxlY3R1cmVyQFRET1kiLC
        JmaXJzdG5hbWUiOiJSb2JlcnQiLCJsYXN0bmFtZSI6IklzYWFjIiwicm9sZXMiOlsiVVNFUiJdLCJpYXQiOjE2OTYwNDEy
        ODEsImV4cCI6MTY5NjEyNzY4MX0.NDoPBqNiQNlxIF8mmnjxMJ_QQfrmVPU6H38Ez1fsg-c",
    "expires_in": 900, "refresh_expires_in": 3600,
    "refresh_token":"eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI2NzFhOWY5My1hYmYxLTRiYmItODg2ZC1lN2UxYjI
    0ZTZkNjYifQ.eyJleHAiOjE2OTYwNTI...”,
    "type": "Bearer"
}
```


```
curl -v -H "Authorization: Bearer <accesstoken>" localhost:8090/elibrary/api/v1/orders/2
```


```
{
    "id": 2,
    "isbn": "abcde1234",
    "expiry": "2023-10-21"
}
```

Bearer Token Authentication

- Spring



https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html

Spring - Implementation

- Configure spring for OAuth 2.0 Resource Server support
- Understand the format and content of your JWT claim
- BearerTokenAuthenticationFilter
- Custom filter (authTokenFilter) to handle other authentication details

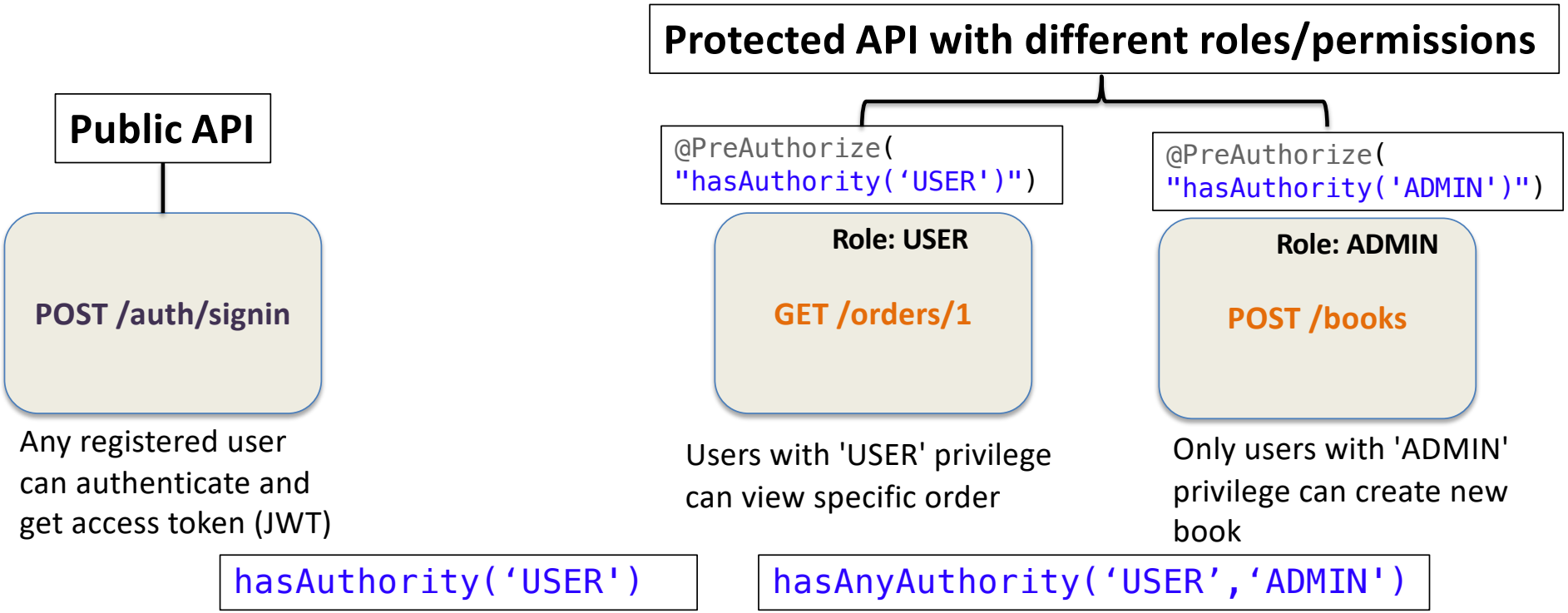
```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf(csrf->csrf.disable());
    http.sessionManagement(session ->
    session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
    http.authorizeHttpRequests(authorize -> authorize.anyRequest().authenticated());
    http.oauth2ResourceServer(oauth2 -> oauth2
        .jwt(jwtconfig -> jwtconfig.jwtAuthenticationConverter(jwt -> RoleConverter(jwt))));

    http.addFilterAfter(authTokenFilter, BearerTokenAuthenticationFilter.class);

    return http.build();
}
```

Authorization (Implementation)

- Spring provides the APIs to perform method-level role-based authorization for REST APIs
  - @PreAuthorize annotation



Authorization - Implementation

- Can do more:
  - Attribute-Based Access Control (ABAC)
  - Using the **Authentication** object + SpEL

```
authentication.getPrincipal() -> authentication.principal
authentication.getDetails() -> authentication.details
```

```
public class UserDetailsImpl implements UserDetails {

private Long userid;
private String firstname;
private String lastname;
private String email;
private String department;    // attribute
private String country;       // attribute
...
```

```
@PreAuthorize(
"hasAuthority('USER') and #country ==
authentication.details.country")
```

GET
countries/{country}/users

Users with 'USER' privilege and that also belong to a certain country can view other users of the same country

eLibrary REST API Endpoints

Resource	API Method	Endpoint (URI Path)	HTTP Method	Allowed Authority/Access
Create/Register a user	createUser	/users	POST	USER
Create a borrow book order for a user	borrowBook	/users/{id}/orders	POST	
Delete a user	deleteUser	/users/{id}	DELETE	
Delete (Return) a book order	returnBook	/orders/{id}	DELETE	
List all borrow orders	getBorrowOrders	/orders	GET	
Details of a borrow order	getBorrowOrder	/orders/{id}	GET	

eLibrary REST API Endpoints

Resource	**API Method**	**Endpoint (URI Path)**	**HTTP Method**	**Allowed Authority/Access**
List of authors	getAuthors	/authors	GET	
Details of each author	getAuthor	/authors/{id}	GET	
Create a new author	createAuthor	/authors	POST	
Update an author	updateAuthor	/authors/{id}	PUT	
List all users	getUsers	/users	GET	
Details of each user	getUser	/users/{id}	GET	
Create a user	createUser	/users	POST	
Update a user	updateUser	/users/{id}	PUT	