Spring Security

Spring Security as a powerful and highly customizable framework for authentication and access control

Introduction

- . The framework that enormously simplifies baking security for Spring applications. Spring Security is the primary choice for implementing application-level security in Spring applications.
- A highly customizable way of implementing authentication, authorization, and protection against common attacks.
- Spring Security is open-source software released under the Apache 2.0 license. https://github.com/spring-proj-ects/spring-security/

Concept

- The framework's philosophy starts with the management of the Spring context. You define beans in the Springcontext to allow the framework to manage them based on configurations you specify.
- Annotations to make these configurations and leave behind the old-fashioned XML configuration style! to do: expose endpoints, wrap methods in transactions, intercept methods in aspects, and so on.
- Apache Shiro (https://shiro.apache.org), We can use as an alternative of Spring Security is

Apache Shiro

- Offers flexibility in configurations and is easy to integrate with Spring and SpringBoot applications.
- Apache Shiro makes, sometimes, a good alternative to the SpringSecurity approach. If you've already worked with Spring Security, you'll find using Apache Shiro easy and comfortable to learn and use.
- It offers its own annotations and design for web applications based on HTTP filters, which are of great simplicity for web applications.
- You can secure more than web applications with Shiro, from smaller command-line applications and mobile applications to large-scale enterprise applications.

Apache Shiro

- It’s powerful enough to use for a wide range of things from authentication and authorization to cryptography and session management.
- Apache Shiro could be too “light” for the needs of your application. SpringSecurity is not only a hammer, but an entire set of tools.
- A larger scale of possibilities and is designed specifically for Spring applications. Moreover, it benefits from a larger community of active developers, and it’s continuously enhanced.

Security Threats

- Broken authentication
- Session fixation
- Cross-site scripting (XSS)
- Cross-site request forgery (CSRF)
- Injections
- Sensitive data exposure
- Lack of method access control
- Using dependencies with known vulnerabilitie

Authentication

- The process in which an application identifies someone trying to use it. When someone or some-thing uses the application, we want to obtain their identity so that further access is granted or not.
- Cases exist in which the access is anonymous, but in most situations,data and actions can be used only by identified requests.
- Once we have the identity of the user, we can process the authorization.

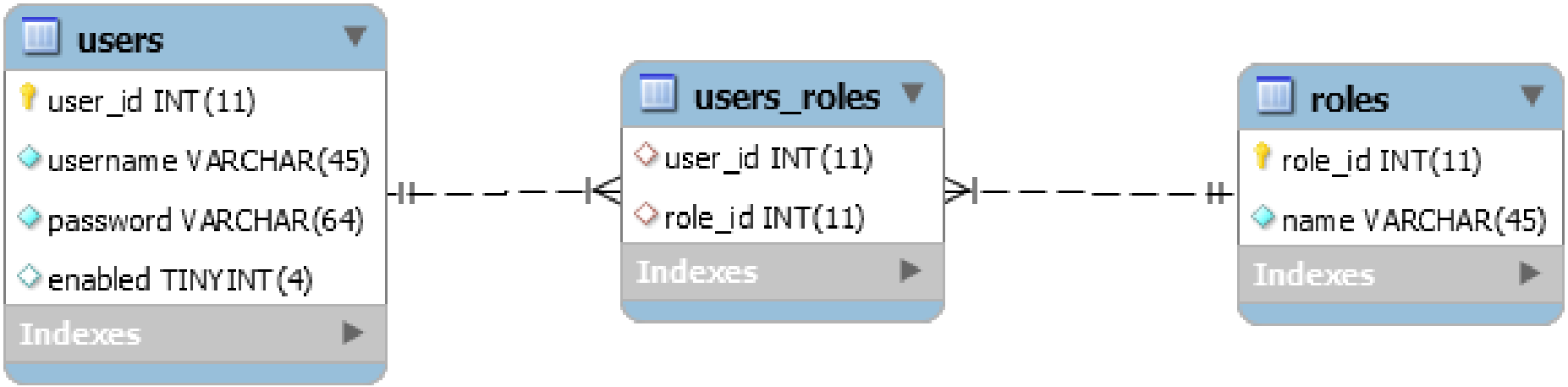
Authorize users based on their roles

- The credentials and roles are stored dynamically in MySQL database. Spring Data JPA with Hibernate is used for the data access layer and Thymeleaf integration with Spring Security is used for the view layer.
- The role USER allows user to view all products; the role CREATOR is permission to create new products; the role EDITOR is for editing products; and the role ADMIN gives all permissions to the users.

Authorize users based on their roles

Username	Roles
prabhat	USER
ajay	CREATOR
jagdeeshwaran	EDITOR
narendra	CREATOR, EDITOR
admin	ADMIN

Authorize users based on their roles



Role.java

```
@Entity
@Table(name = "roles")
public class Role {
    @Id
    @Column(name = "role_id")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;

    private String name
……..
}
```

Users.java

```
@Entity
@Table(name = "users")
public class Users {
    @Id
    @Column(name = "user_id")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String username, password;
    private boolean enabled;
    @ManyToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JoinTable(
            name = "users_roles",
            joinColumns = @JoinColumn(name = "user_id"),
            inverseJoinColumns = @JoinColumn(name = "role_id")
            )
    private Set<Role> roles = new HashSet<>();
……………………………………………………………………………
}
```

UsersRepository.java

```
public interface UsersRepository extends JpaRepository<User, Long> {


    @Query("SELECT u FROM Users u WHERE u.username = :username")
    public User getUserByUsername(@Param("username") String username);
}
```

MyUserDetails.java

```
public class MyUserDetails implements UserDetails {
    private User user;
    public MyUserDetails(User user) {
        this.user = user;
    }
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        Set<Role> roles = user.getRoles();
        List<SimpleGrantedAuthority> authorities = new ArrayList<>();
        for (Role role : roles) {
            authorities.add(new SimpleGrantedAuthority(role.getName()));
        }
        return authorities;
    }
    @Override
    public String getPassword() {
        return user.getPassword();
    }
    @Override
    public String getUsername() {
        return user.getUsername();
    }
    @Override
    public boolean isAccountNonExpired() {
        return true;
    }
    @Override
    public boolean isAccountNonLocked() {
        return true;
    }
    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }
    @Override
    public boolean isEnabled() {
        return user.isEnabled();
    }}
```

UserDetailsServiceImpl .java

```
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    private UserRepository userRepository;
    @Override
    public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {
        User user = userRepository.getUserByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException("Could not find user");
        }
        return new MyUserDetails(user);
    }
}
```

WebSecurityConfig.java

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {
    @Bean
    public UserDetailsService userDetailsService() {
        return new UserDetailsServiceImpl();
    }
    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
    @Bean
    public DaoAuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authProvider = new
DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userDetailsService());
        authProvider.setPasswordEncoder(passwordEncoder());
        return authProvider;
    }
@Override
    protected void configure(AuthenticationManagerBuilder
auth) throws Exception {
        auth.authenticationProvider(authenticationProvider());
    }
    @Override
    protected void configure(HttpSecurity http) throws
Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin().permitAll()
            .and()
            .logout().permitAll();
    }
}
```

Authorization

- Authorization is the process of establishing if an authenticated caller has the privileges to use specific functionality and data.
- In a mobile banking application, most of the authenticated users can transfer money, but only from their account.

Session Fixation

- Session fixation vulnerability is a more specific, high severity weakness of a web application. It could permit an attacker to impersonate a valid user by the reuse of apreviously generated session ID.
- This vulnerability could appear if, during the authentication process, the web application doesn’t assign a new session ID, and this could make possible the reuse of existing session IDs.
- Exploiting this vulnerability consists of obtaining a valid session ID and making the intended victim’s browser use it.

Cross-site scripting

- XSS, allows the injection of client-side scripts into web services exposed by the server, thereby permitting other users to run them.
- Before being used, or even stored, the request should be properly “sanitized” to avoid undesired executions of foreign scripts.
- The potential impact could relate to account impersonation (for example, combined with session fixation) or to participation in distributed attacks like DDoS

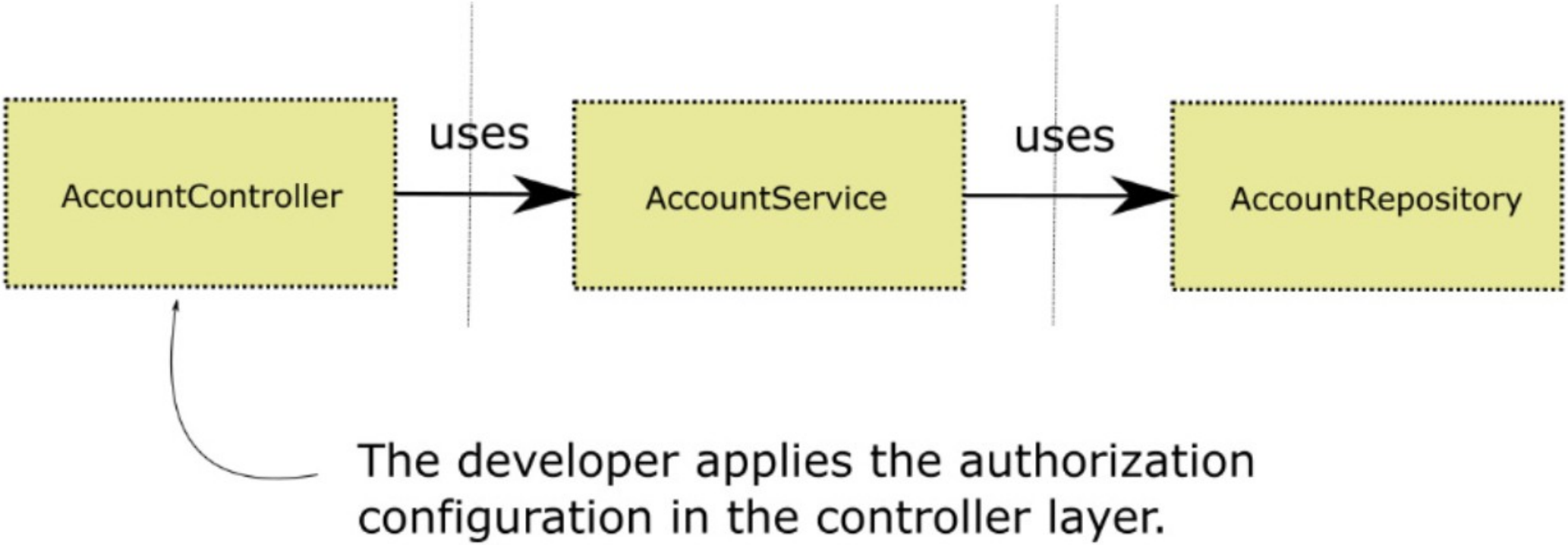
Cross-site request forgery (CSRF)

- CSRF attacks assume a URL that calls an action on a specific server can be extracted and reused from outside of the application.
- A server trusts the execution without doing any check on the origin of the request, one could execute it from any other place.
- An attacker could make a user execute undesired actions on a server by hiding the actions. Usually, with this vulnerability, the attacker targets actions that change data in the system.
- use tokens to identify the request or use Cross-Origin Resource Sharing (CORS) limitations. With this method,you validate the origin of the request.

Injection vulnerabilities

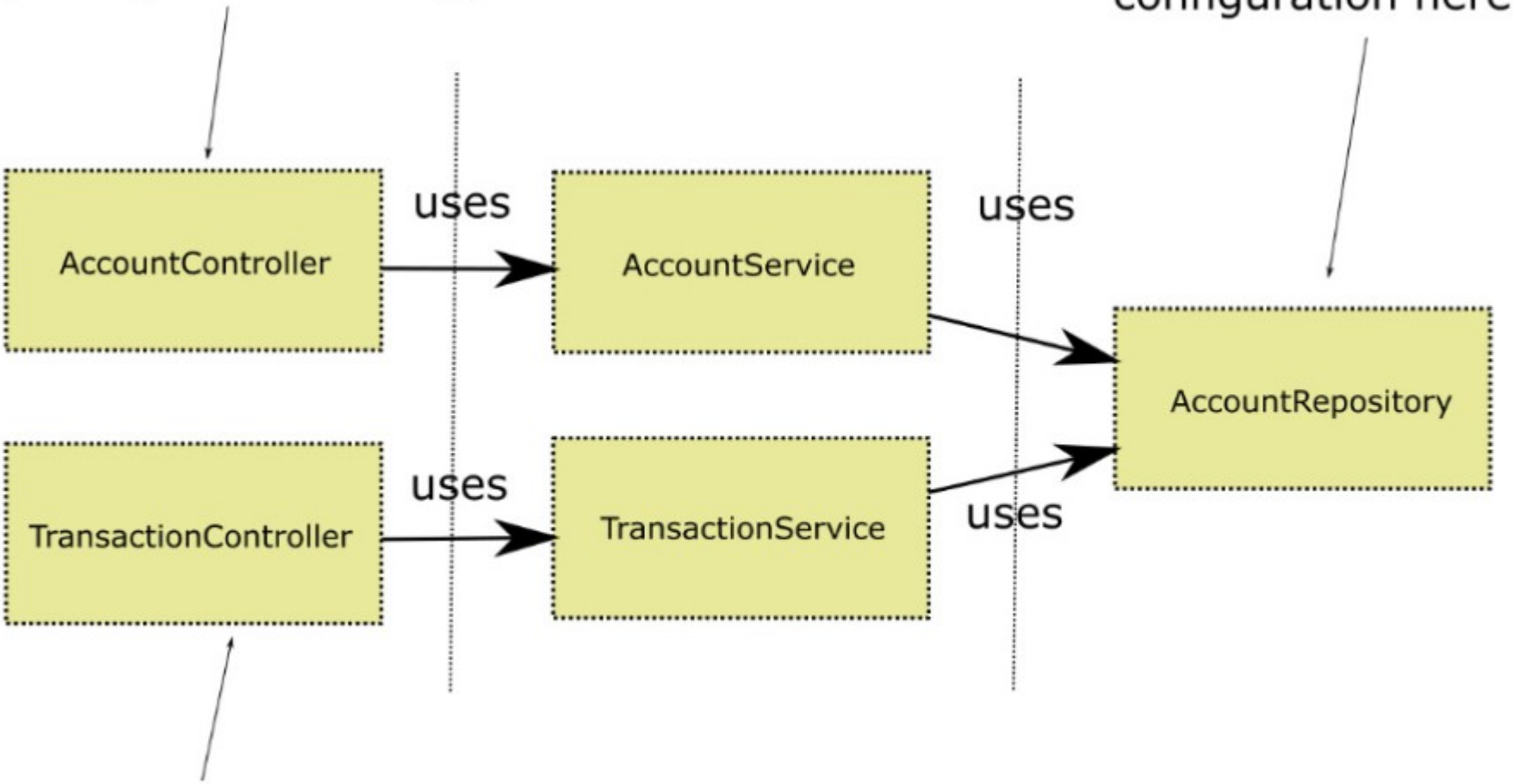
- An injection attack, the attacker,employing a vulnerability, introduces specific data into the system.
- The purpose is to harm the system, change data in an unwanted way, or retrieve data that’s not meant to be accessed by them.
- Many types of injection attacks exist. XSS can be considered an injection vulnerability.
- Injection attacks inject a client-side script with the means of harming the system somehow. Other examples could be SQL injection, XPath injection, OS command injection, LDAP injection, and the list continues.

Architecture

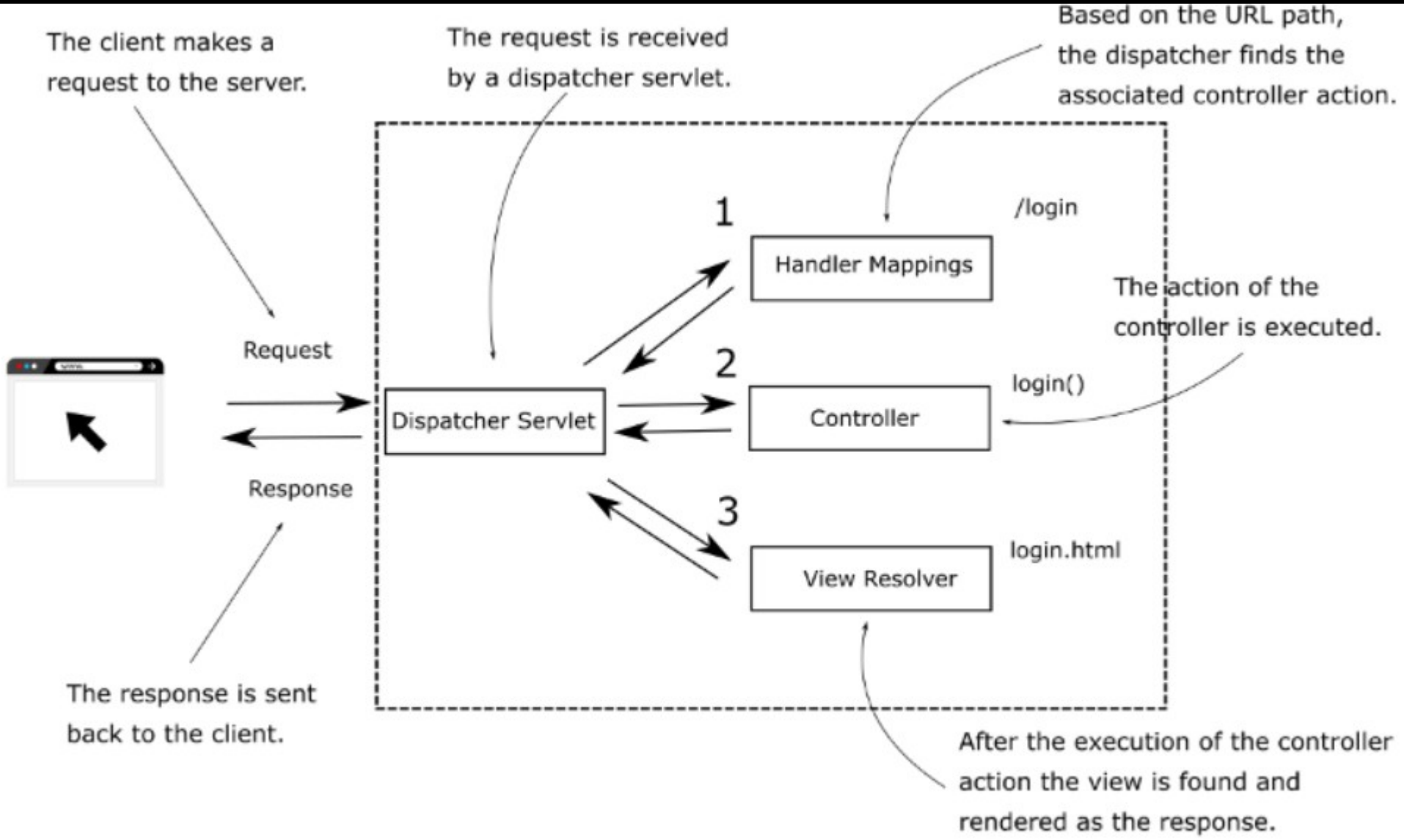


1. The authorization configuration regarding account is applied here.

3. So better move the configuration here.



2. But we also need to apply authorization configuration regarding accounts here now.


The client makes a request to the server.
The request is received by a dispatcher servlet.
Based on the URL path, the dispatcher finds the associated controller action.
1
/login
Handler Mappings
The action of the controller is executed.
Request
2
login()
Dispatcher Servlet
Controller
Response
3
login.html
View Resolver
The response is sent back to the client.
After the execution of the controller action the view is found and rendered as the response.

OAuth2 flow

- A solution to avoid resending credentials for each of the requests to the back-end and store them on the client-side.
- The OAuth2 flow offers a better way to implement authentication and authorization in this case.
- The authorization server and the resource server. The purpose of the authorization server is to authorize the user and provide them with a token that specifies, among other things, a set of privileges that can be used.
- The part of the backend implementing the functionality is called the resource server. The endpoints that can be called are considered protected resources.
- Based on the obtained token, after accomplishing the authorization, a call on a resource will be permitted or rejected.

OAuth2 flow

- The user accesses a use case in the application (also known as the client). The application needs to call a resource in the backend.
- To be able to call the resource, the application first has to obtain an access token, so it calls the authorization server to get the token.
- In the request, it sends the user's credentials or a refresh token in certain cases.
- If the credentials or the refresh token are correct, the authorization server returns a (new) access token to the client.
- The access token is used in the header of the request to the resource server when calling the needed resources.

OAuth2 flow

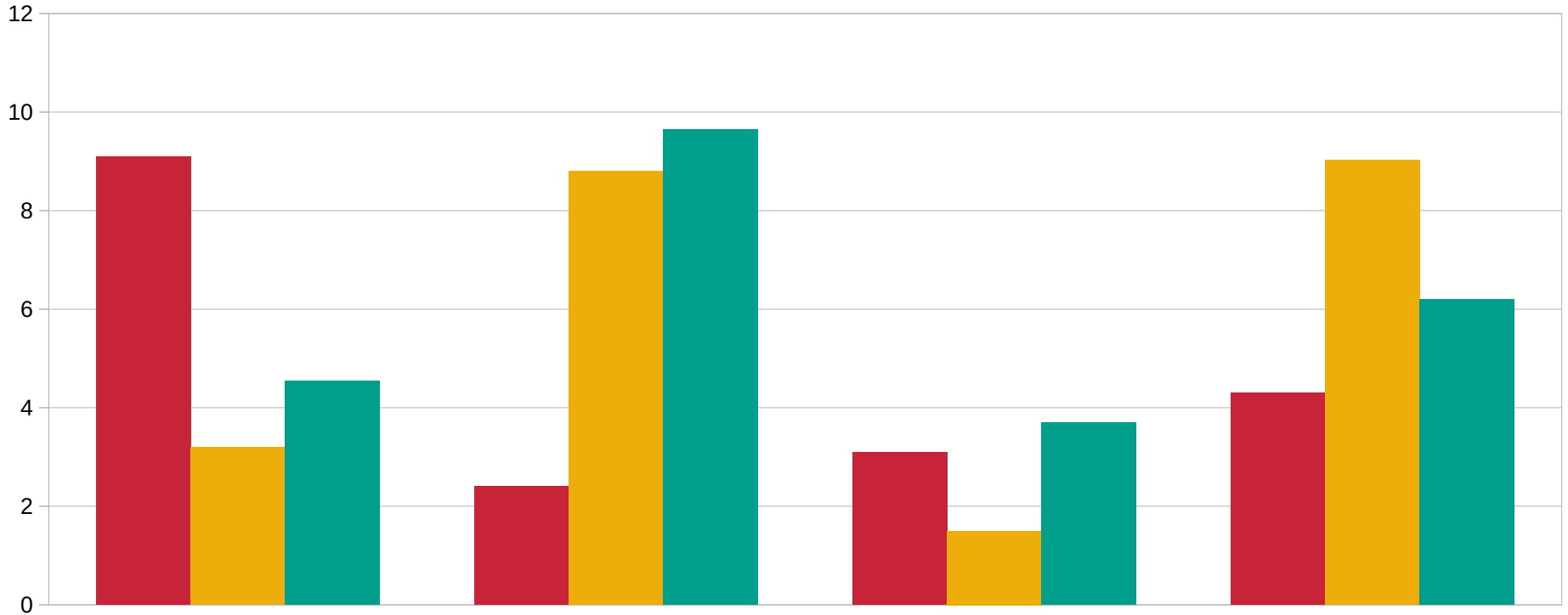
- As a visitor, you first visit the front desk, where you receive an access card after identifying yourself. The access card can open some of the doors, but not necessarily all. Based on your identity, you can access exactly the doors that you're allowed to and no more. The same happens with an access token. After the authentication, the caller is provided with a token, and based on that, they can access the resources for which they have privileges. A token has a determined lifetime, usually being short-lived. Depending on the implementation, when a token expires, a new authorization should be made either with a refresh token or with user credentials. If needed, the token can be dis-qualified by the server earlier than its expiration time.

OAuth2 flow

- The client doesn’t have to store the user’s credentials. The access token and,eventually, the refresh token are the only access details needed to be saved. The application doesn’t expose the user’s credentials that are often on the network. If someone intercepts a token, you can disqualify the token without needing to invalidate the user’s credentials. A token can be used by a third entity to access resources on the user’s behalf,without having to impersonate the user. An attacker could steal the token in this case. But, because the token usually has a limited lifespan, the time frame in which one can use this vulnerability is limited.

Using API keys, cryptographic signatures, and IP whitelisting to secure requests

- Using API keys, cryptographic signatures, and IP whitelisting to secure requests In certain cases, you don’t need a username and a password to authenticate and authorize a caller, but you still want to make sure that nobody altered the exchanged messages. You might need this approach when the requests are made between two backend components. Sometimes you’d like to make sure that the messages between them are validated somehow.


12
10
8
6
4
2
0