

Spring Boot API features

write spring boot api and add below features

1. Rate limiting
2. CORS
3. SQL & NoSQL injection
4. Firewalls
5. VPN's
6. CSRF
7. XSS (Cros-site scripting)

add above features and explain each features with example code

Got it! Here's a compact, working-style Spring Boot API setup that layers in each security feature you asked for—**rate limiting, CORS, SQL & NoSQL injection protections, firewalls, VPN/IP allowlisting, CSRF, and XSS**—with focused code snippets and quick explanations.

0) Minimal project setup

pom.xml (key deps)

xml

```
<dependencies>
  <!-- Web + Security -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>

  <!-- SQL (H2) + JPA -->
  <dependency>
```

```

    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>

<!-- NoSQL (MongoDB) -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>

<!-- Rate limiting -->
<dependency>
    <groupId>com.github.vladimir-bukhtoyarov</groupId>
    <artifactId>bucket4j-core</artifactId>
    <version>8.10.1</version>
</dependency>

<!-- XSS sanitize -->
<dependency>
    <groupId>org.jsoup</groupId>
    <artifactId>jsoup</artifactId>
    <version>1.17.2</version>
</dependency>
</dependencies>

```

application.yml (demo)

```

yaml

server:
  port: 8080
spring:
  h2:
    console:
      enabled: true
  datasource:
    url: jdbc:h2:mem:testdb;MODE=PostgreSQL
    driverClassName: org.h2.Driver

```

```
username: sa
password:
jpa:
  hibernate:
    ddl-auto: update
  show-sql: false
data:
  mongodb:
    database: demo
    uri: mongodb://localhost:27017/demo
```

1) Demo API

java

```
// com.example.secureapi/SecureApiApplication.java
@SpringBootApplication
public class SecureApiApplication {
    public static void main(String[] args) {
        SpringApplication.run(SecureApiApplication.class, args);
    }
}
```

java

```
// com.example.secureapi/api/ItemController.java
@RestController
@RequestMapping("/api/items")
public class ItemController {
    private final ItemService service;

    public ItemController(ItemService service) { this.service = service; }

    @GetMapping
    public List<ItemDto> list(@RequestParam(required = false) String q) {
        return service.search(q);
    }

    @PostMapping
    public ItemDto create(@RequestBody @Valid CreateItemRequest req) {
```

```
    return service.create(req);  
}  
}
```

java

```
// com.example.secureapi/api/dto/CreateItemRequest.java  
public class CreateItemRequest {  
    @NotBlank  
    private String name;  
  
    @Size(max = 1000)  
    private String description;  
  
    // getters/setters  
}
```

java

```
// com.example.secureapi/api/dto/ItemDto.java  
public class ItemDto {  
    private Long id;  
    private String name;  
    private String description;  
    // getters/setters, constructors  
}
```

2) SQL injection (safe by default with JPA)

Why: SQL injection happens when untrusted input is concatenated into queries. Use **parameter binding** (Spring Data/JPA does this automatically).

java

```
// com.example.secureapi/sql/Item.java  
@Entity  
public class Item {  
    @Id @GeneratedValue  
    private Long id;  
    private String name;
```

```
@Column(length = 1000)
private String description;
// getters/setters
}
```

java

```
// com.example.secureapi/sql/ItemRepository.java
public interface ItemRepository extends JpaRepository<Item, Long> {
    // SAFE: binds param properly (no string concatenation)
    List<Item> findByNameContainingIgnoreCase(String name);

    // SAFE custom JPQL with :param
    @Query("select i from Item i where lower(i.description) like lower(concat('%', :kw, '%'))")
    List<Item> searchByDescription(@Param("kw") String kw);
}
```

❌ Avoid (unsafe):

java

```
// Don't do this
@Query(value = "select * from item where name like '%" + userInput + "%'", nativeQuery =
true)
```

3) NoSQL injection (MongoDB) protections

Why: Operators like `$where` can execute JS, and operator injection (`{"$ne": null}`) can alter queries. Build queries with `Criteria` and validate/whitelist fields.

java

```
// com.example.secureapi/nosql/ItemDoc.java
@Document("items")
public class ItemDoc {
    @Id private String id;
    private String name;
    private String description;
}
```

```
// getters/setters
}
```

java

```
// com.example.secureapi/nosql/ItemDocRepository.java
public interface ItemDocRepository extends MongoRepository<ItemDoc, String> {
    // Derived queries are parameterized internally (safe)
    List<ItemDoc> findByNameRegex(String regex);
}
```

java

```
// com.example.secureapi/nosql/ItemDocService.java
@Service
public class ItemDocService {
    private final MongoTemplate mongo;

    public ItemDocService(MongoTemplate mongo) { this.mongo = mongo; }

    public List<ItemDoc> safeSearch(String input) {
        // Escape regex metacharacters to prevent operator tricks
        String safe = Pattern.quote(input == null ? "" : input);
        Query q = new Query(Criteria.where("name").regex(safe, "i"));
        // NEVER use $where or raw JSON concatenation
        return mongo.find(q, ItemDoc.class);
    }
}
```

4) XSS (Cross-Site Scripting)

Why: If your API echoes HTML provided by users (e.g., descriptions) and a web client renders it, sanitize before storing or before returning to HTML contexts. Also set restrictive headers.

java

```
// com.example.secureapi/xss/XssUtil.java
public final class XssUtil {
    private XssUtil() {}
}
```

```

public static String sanitize(String html) {
    // Allow only very basic safe tags; adjust as needed
    return Jsoup.clean(html, Safelist.basic());
}
}

```

Use it in service:

```

java

// com.example.secureapi/service/ItemService.java
@Service
public class ItemService {
    private final ItemRepository repo;
    private final ItemDocService docService;

    public ItemService(ItemRepository repo, ItemDocService docService) {
        this.repo = repo; this.docService = docService;
    }

    @Transactional
    public ItemDto create(CreateItemRequest req) {
        Item item = new Item();
        item.setName(req.getName().trim());
        item.setDescription(XssUtil.sanitize(req.getDescription()));
        Item saved = repo.save(item);
        return toDto(saved);
    }

    public List<ItemDto> search(String q) {
        List<Item> results = (q == null || q.isBlank())
            ? repo.findAll()
            : repo.findByNameContainingIgnoreCase(q.trim());
        return results.stream().map(this::toDto).toList();
    }

    private ItemDto toDto(Item i) {
        ItemDto dto = new ItemDto();
        dto.setId(i.getId());
        dto.setName(i.getName());
        dto.setDescription(i.getDescription());
        return dto;
    }
}

```

```
}  
}
```

Secure response headers (CSP, etc.) are configured in Security (see section 7).

5) CORS

Why: Control which front-ends can call your API from browsers.

java

```
// com.example.secureapi/security/CorsConfig.java  
@Configuration  
public class CorsConfig {  
    @Bean  
    public CorsConfigurationSource corsConfigurationSource() {  
        CorsConfiguration cfg = new CorsConfiguration();  
        cfg.setAllowedOrigins(List.of("https://app.example.com")); // exact origins  
        cfg.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));  
        cfg.setAllowedHeaders(List.of("Content-Type", "Authorization", "X-XSRF-TOKEN"));  
        cfg.setAllowCredentials(true);  
        cfg.setMaxAge(3600L);  
  
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();  
        source.registerCorsConfiguration("/**", cfg);  
        return source;  
    }  
}
```

6) Rate Limiting (Bucket4j per IP)

Why: Throttle abusive clients (DoS/hammering).

java

```
// com.example.secureapi/rate/RateLimitFilter.java  
@Component  
public class RateLimitFilter extends OncePerRequestFilter {  
  
    private final Map<String, Bucket> buckets = new ConcurrentHashMap<>();  

```



```

private Bucket newBucket() {
    // e.g., 100 requests per minute burst
    Refill refill = Refill.greedy(100, Duration.ofMinutes(1));
    Bandwidth limit = Bandwidth.classic(100, refill);
    return Bucket4j.builder().addLimit(limit).build();
}

private Bucket resolveBucket(String key) {
    return buckets.computeIfAbsent(key, k -> newBucket());
}

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

    String ip = Optional.ofNullable(request.getHeader("X-Forwarded-For"))
        .map(v -> v.split(",")[0].trim())
        .orElse(request.getRemoteAddr());

    Bucket bucket = resolveBucket(ip);
    if (bucket.tryConsume(1)) {
        filterChain.doFilter(request, response);
    } else {
        response.setStatus(HttpStatus.TOO_MANY_REQUESTS.value());
        response.getWriter().write("Too many requests");
    }
}
}

```

Register it early in the chain:

```

java

// in SecurityConfig (next section)
http.addFilterBefore(rateLimitFilter, UsernamePasswordAuthenticationFilter.class);

```

7) Spring Security: CSRF, headers (XSS/CSP), firewall

java

// com.example.secureapi/security/SecurityConfig.java

@Configuration

@EnableWebSecurity

public class SecurityConfig {

private final RateLimitFilter rateLimitFilter;

public SecurityConfig(RateLimitFilter rateLimitFilter) {

this.rateLimitFilter = rateLimitFilter;

}

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.cors(Customizer.withDefaults())

// For browser-based sessions, keep CSRF ON; for pure stateless token APIs you may disable.

.csrf(csrf -> csrf

.csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse()) *// sends XSRF-*

TOKEN cookie

)

.authorizeHttpRequests(auth -> auth

.requestMatchers("/h2-console/**").permitAll()

.anyRequest().permitAll()

)

// Security headers: XSS/Clickjacking/ReferrerPolicy/CSP/etc.

.headers(headers -> headers

.frameOptions(frame -> frame.sameOrigin()) *// for H2 console*

.contentSecurityPolicy(csp -> csp

.policyDirectives("default-src 'none'; " +

"img-src 'self'; " +

"script-src 'self'; " +

"style-src 'self'; " +

"connect-src 'self' https://app.example.com; " +

"frame-ancestors 'none';"))

.referrerPolicy(ref ->

ref.policy(ReferrerPolicyHeaderWriter.ReferrerPolicy.NO_REFERRER))

.xssProtection(x -> x.block(true)) *// (legacy header; CSP is primary)*

);

http.addFilterBefore(rateLimitFilter, UsernamePasswordAuthenticationFilter.class);

return http.build();

```

}

// Application-level request firewall (blocks suspicious requests)
@Bean
public HttpFirewall httpFirewall() {
    StrictHttpFirewall fw = new StrictHttpFirewall();
    fw.setAllowUrlEncodedDoubleSlash(false);
    fw.setAllowUrlEncodedPercent(false);
    fw.setAllowedHttpMethods(Set.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));
    return fw;
}

@Bean
public WebSecurityCustomizer webSecurityCustomizer(HttpFirewall firewall) {
    return web -> web.httpFirewall(firewall);
}
}

```

CSRF usage tip (browser clients):

- Client reads `XSRF-TOKEN` cookie and sends it back in header `X-XSRF-TOKEN` on mutating requests (POST/PUT/DELETE). Spring validates it.

8) Firewalls (app-layer) & network

What this covers in code: `StrictHttpFirewall` blocks encoded attacks, invalid methods, etc.

What you should also do outside the app: configure cloud/network firewalls (AWS SG/ACL, Azure NSG) to restrict ports and IP ranges.

9) VPNs / IP allowlisting / mTLS

Concept: Put the API behind a VPN (WireGuard/OpenVPN/Cloud/VPC). At the app layer, you can also **allowlist** IPs or **require** mTLS for private APIs.

(A) Simple IP allowlist filter

```
java
```

```
// com.example.secureapi/ip/IpAllowlistFilter.java
@Component
@Order(1) // run very early
public class IpAllowlistFilter extends OncePerRequestFilter {

    private static final Set<String> ALLOWED = Set.of("10.0.0.5", "203.0.113.10"); // example

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException, IOException {
        String ip = Optional.ofNullable(request.getHeader("X-Forwarded-For"))
            .map(v -> v.split(",")[0].trim())
            .orElse(request.getRemoteAddr());
        if (!ALLOWED.contains(ip)) {
            response.setStatus(HttpStatus.FORBIDDEN.value());
            response.getWriter().write("Forbidden (IP not allowed)");
            return;
        }
        filterChain.doFilter(request, response);
    }
}
```

Register it **before** other filters (e.g., before rate limiting) if you want to block early.

(B) mTLS (server-side config)

In `application.yml`:

```
yaml

server:
  ssl:
    enabled: true
    key-store: classpath:server-keystore.p12
    key-store-password: changeit
    key-store-type: PKCS12
    client-auth: need
    trust-store: classpath:server-truststore.p12
    trust-store-password: changeit
    trust-store-type: PKCS12
```

Clients must present a trusted certificate—perfect for VPN/internal APIs.

10) Putting it together: end-to-end flow

- **CORS** only lets your approved front-end origin call the API from browsers.
- **Rate limiting** throttles each client IP.
- **Firewall (StrictHttpFirewall)** drops suspicious/encoded/method-abuse requests.
- **VPN/IP allowlist/mTLS** restricts *who* can even reach the API.
- **CSRF** protects browser users by requiring the `X-XSRF-TOKEN` header on state-changing requests.
- **SQL/NoSQL injection** protections are enforced by parameterized queries & Criteria API (no string concat).
- **XSS** is mitigated by sanitizing user input and enforcing CSP headers.

Quick test notes

- For POST `/api/items`, send JSON:

json

```
{ "name": "Hello", "description": "<b>bold</b><script>alert(1)</script>" }
```

Stored description becomes sanitized HTML (no `<script>`), and CSP prevents inline scripts anyway.

- From a browser client, include CSRF header:

csharp

```
X-XSRF-TOKEN: <value from XSRF-TOKEN cookie>
```

- Hit the endpoint >100 times/min from same IP → `429 Too Many Requests`.

If you want, I can wrap this into a small Git repo layout (packages + files) or adapt it to **JWT stateless APIs** (CSRF off, CORS + CSP still on, and rate limit by API key).

2/2

Frontend use React

I don't have any ui and api code, do from scratch, you

think i don't have any configurations also so please give UI and API code from scratch step by step, please give code in chat

Awesome—let's build a **from-scratch Spring Boot API + React UI** that includes:

- Rate limiting
- CORS
- SQL & NoSQL injection protections
- App-layer firewall
- VPN/IP allowlisting (switchable)
- CSRF
- XSS protections

I'll give you the exact files to create. Copy-paste them as is and you'll have a working demo.

Part A — Spring Boot API (Java 17+, Maven)

1) Create project

Folder: `secure-api/`

`pom.xml`

xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.3.2</version>
    <relativePath/>
  </parent>
```

```
<groupId>com.example</groupId>
<artifactId>secure-api</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>secure-api</name>
<description>Spring Boot API with Security features</description>

<properties>
  <java.version>17</java.version>
</properties>

<dependencies>
  <!-- Web + Security + Validation -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
  </dependency>

  <!-- SQL (H2) + JPA -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>

  <!-- Mongo (NoSQL) -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
  </dependency>
```

```

<!-- Rate limiting -->
<dependency>
  <groupId>com.github.vladimir-bukhtoyarov</groupId>
  <artifactId>bucket4j-core</artifactId>
  <version>8.10.1</version>
</dependency>

<!-- XSS sanitize -->
<dependency>
  <groupId>org.jsoup</groupId>
  <artifactId>jsoup</artifactId>
  <version>1.17.2</version>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.11.0</version>
      <configuration>
        <release>17</release>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>

```

src/main/resources/application.yml

yaml


```

server:
  port: 8080

spring:
  h2:
    console:
      enabled: true
      path: /h2-console
  datasource:
    url: jdbc:h2:mem:testdb;MODE=PostgreSQL
    driverClassName: org.h2.Driver
    username: sa
    password:
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: false

data:
  mongodb:
    # If you don't have Mongo running, leave as is. Endpoints using Mongo will 500 if hit.
    uri: mongodb://localhost:27017/demo
    database: demo

# Toggle IP allowlisting here
app:
  ip-allowlist:
    enabled: false
    addresses: 127.0.0.1,0:0:0:0:0:0:1

# Allowed front-end origin
cors:
  allowed-origins:
    - http://localhost:5173

```

```
src/main/java/com/example/secureapi/SecureApiApplication.java
```

```

java

package com.example.secureapi;

import org.springframework.boot.SpringApplication;

```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SecureApiApplication {
    public static void main(String[] args) {
        SpringApplication.run(SecureApiApplication.class, args);
    }
}
```

2) Domain, DTO, Repo, Service

src/main/java/com/example/secureapi/sql/Item.java

```
java

package com.example.secureapi.sql;

import jakarta.persistence.*;

@Entity
public class Item {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(length = 1000)
    private String description;

    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }
}
```

src/main/java/com/example/secureapi/sql/ItemRepository.java

java

```
package com.example.secureapi.sql;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

import java.util.List;

public interface ItemRepository extends JpaRepository<Item, Long> {
    List<Item> findByNameContainingIgnoreCase(String name);

    @Query("select i from Item i where lower(i.description) like lower(concat('%', :kw, '%'))")
    List<Item> searchByDescription(@Param("kw") String kw);
}
```

src/main/java/com/example/secureapi/api/dto/CreateItemRequest.java

a

java

```
package com.example.secureapi.api.dto;

import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;

public class CreateItemRequest {
    @NotBlank
    private String name;

    @Size(max = 1000)
    private String description;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }
}
```

```
src/main/java/com/example/secureapi/api/dto/ItemDto.java
```

```
java
```

```
package com.example.secureapi.api.dto;

public class ItemDto {
    private Long id;
    private String name;
    private String description;

    public ItemDto() {}
    public ItemDto(Long id, String name, String description) {
        this.id = id; this.name = name; this.description = description;
    }

    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }
}
```

```
src/main/java/com/example/secureapi/xss/XssUtil.java
```

```
java
```

```
package com.example.secureapi.xss;

import org.jsoup.Jsoup;
import org.jsoup.safety.Safelist;

public final class XssUtil {
    private XssUtil() {}
    public static String sanitize(String html) {
        if (html == null) return null;
        return Jsoup.clean(html, Safelist.basic());
    }
}
```

```
java
```

```
package com.example.secureapi.service;

import com.example.secureapi.api.dto.CreateItemRequest;
import com.example.secureapi.api.dto.ItemDto;
import com.example.secureapi.sql.Item;
import com.example.secureapi.sql.ItemRepository;
import com.example.secureapi.xss.XssUtil;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service
public class ItemService {
    private final ItemRepository repo;

    public ItemService(ItemRepository repo) {
        this.repo = repo;
    }

    @Transactional
    public ItemDto create(CreateItemRequest req) {
        Item i = new Item();
        i.setName(req.getName().trim());
        i.setDescription(XssUtil.sanitize(req.getDescription()));
        Item saved = repo.save(i);
        return new ItemDto(saved.getId(), saved.getName(), saved.getDescription());
    }

    public List<ItemDto> search(String q) {
        List<Item> list = (q == null || q.isBlank())
            ? repo.findAll()
            : repo.findByNameContainingIgnoreCase(q.trim());
        return list.stream().map(it -> new ItemDto(it.getId(), it.getName(),
it.getDescription())).toList();
    }
}
```

3) NoSQL (safe query via Criteria)

src/main/java/com/example/secureapi/nosql/ItemDoc.java

java

```
package com.example.secureapi.nosql;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document("items")
public class ItemDoc {
    @Id
    private String id;
    private String name;
    private String description;

    public String getId() { return id; }
    public void setId(String id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }
}
```

src/main/java/com/example/secureapi/nosql/ItemDocService.java

java

```
package com.example.secureapi.nosql;

import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.query.Criteria;
import org.springframework.data.mongodb.core.query.Query;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.regex.Pattern;

@Service
```

```

public class ItemDocService {
    private final MongoTemplate mongo;

    public ItemDocService(MongoTemplate mongo) {
        this.mongo = mongo;
    }

    public List<ItemDoc> safeSearchByName(String input) {
        String safe = Pattern.quote(input == null ? "" : input);
        Query q = new Query(Criteria.where("name").regex(safe, "i"));
        return mongo.find(q, ItemDoc.class);
    }
}

```

(This demonstrates **NoSQL injection protection**: we escape regex & avoid `$where` / raw JSON concatenation.)

4) Controllers (API + CSRF bootstrap)

src/main/java/com/example/secureapi/api/ItemController.java

```

java

package com.example.secureapi.api;

import com.example.secureapi.api.dto.CreateItemRequest;
import com.example.secureapi.api.dto.ItemDto;
import com.example.secureapi.nosql.ItemDoc;
import com.example.secureapi.nosql.ItemDocService;
import com.example.secureapi.service.ItemService;
import jakarta.validation.Valid;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api")
public class ItemController {
    private final ItemService service;
    private final ItemDocService mongoService;
}

```

```

public ItemController(ItemService service, ItemDocService mongoService) {
    this.service = service; this.mongoService = mongoService;
}

@GetMapping("/items")
public List<ItemDto> list(@RequestParam(required = false) String q) {
    return service.search(q);
}

@PostMapping("/items")
public ItemDto create(@RequestBody @Valid CreateItemRequest req) {
    return service.create(req);
}

@GetMapping("/mongo/items")
public List<ItemDoc> mongoSearch(@RequestParam(required = false) String q) {
    return mongoService.safeSearchByName(q);
}

@GetMapping("/health")
public String health() { return "OK"; }
}

```

src/main/java/com/example/secureapi/api/CsrfController.java

```

java

package com.example.secureapi.api;

import org.springframework.security.web.csrf.CsrfToken;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import jakarta.servlet.http.HttpServletRequest;
import java.util.Map;

@RestController
public class CsrfController {
    @GetMapping("/api/csrf")
    public Map<String, String> csrf(HttpServletRequest request) {
        CsrfToken token = (CsrfToken) request.getAttribute(CsrfToken.class.getName());
        // CookieCsrfTokenRepository will also write the cookie
    }
}

```



```
    return Map.of("token", token.getToken());
}
}
```

5) Security: CORS, CSRF, headers (CSP), firewall, rate-limit, IP allowlist

src/main/java/com/example/secureapi/security/CorsConfig.java

```
java

package com.example.secureapi.security;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.CorsConfigurationSource;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;

import java.util.List;

@Configuration
public class CorsConfig {

    @Value("${cors.allowed-origins[0]:http://localhost:5173}")
    private String allowedOrigin;

    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration cfg = new CorsConfiguration();
        cfg.setAllowedOrigins(List.of(allowedOrigin));
        cfg.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));
        cfg.setAllowedHeaders(List.of("Content-Type", "Authorization", "X-XSRF-TOKEN"));
        cfg.setAllowCredentials(true);
        cfg.setMaxAge(3600L);

        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", cfg);
        return source;
    }
}
```

```
}  
}
```

src/main/java/com/example/secureapi/rate/RateLimitFilter.java

java

```
package com.example.secureapi.rate;  
  
import io.github.bucket4j.Bandwidth;  
import io.github.bucket4j.Bucket;  
import io.github.bucket4j.Bucket4j;  
import io.github.bucket4j.Refill;  
import jakarta.servlet.FilterChain;  
import jakarta.servlet.ServletException;  
import jakarta.servlet.http.HttpServletRequest;  
import jakarta.servlet.http.HttpServletResponse;  
import org.springframework.http.HttpStatus;  
import org.springframework.stereotype.Component;  
import org.springframework.web.filter.OncePerRequestFilter;  
  
import java.io.IOException;  
import java.time.Duration;  
import java.util.Map;  
import java.util.Optional;  
import java.util.concurrent.ConcurrentHashMap;  
  
@Component  
public class RateLimitFilter extends OncePerRequestFilter {  
    private final Map<String, Bucket> buckets = new ConcurrentHashMap<>();  
  
    private Bucket newBucket() {  
        Refill refill = Refill.greedy(100, Duration.ofMinutes(1)); // 100 req/min  
        Bandwidth limit = Bandwidth.classic(100, refill);  
        return Bucket4j.builder().addLimit(limit).build();  
    }  
  
    private Bucket resolveBucket(String key) {  
        return buckets.computeIfAbsent(key, k -> newBucket());  
    }  
  
    @Override
```

```

protected void doFilterInternal(HttpServletRequest request,
                                HttpServletResponse response,
                                FilterChain filterChain)
    throws ServletException, IOException {
    String ip = Optional.ofNullable(request.getHeader("X-Forwarded-For"))
        .map(v -> v.split(",")[0].trim())
        .orElse(request.getRemoteAddr());

    if (resolveBucket(ip).tryConsume(1)) {
        filterChain.doFilter(request, response);
    } else {
        response.setStatus(HttpStatus.TOO_MANY_REQUESTS.value());
        response.getWriter().write("Too many requests");
    }
}
}

```

src/main/java/com/example/secureapi/security/IpAllowlistProperties.java

```

java

package com.example.secureapi.security;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;

import java.util.Arrays;
import java.util.List;

@Configuration
@ConfigurationProperties(prefix = "app.ip-allowlist")
public class IpAllowlistProperties {
    private boolean enabled = false;
    private String addresses = "";

    public boolean isEnabled() { return enabled; }
    public void setEnabled(boolean enabled) { this.enabled = enabled; }

    public String getAddresses() { return addresses; }
    public void setAddresses(String addresses) { this.addresses = addresses; }
}

```

```

public List<String> parsed() {
    if (addresses == null || addresses.isBlank()) return List.of();
    return Arrays.stream(addresses.split(","))
        .map(String::trim).filter(s -> !s.isBlank()).toList();
}
}

```

src/main/java/com/example/secureapi/security/IpAllowlistFilter.java

va

java

```

package com.example.secureapi.security;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.core.annotation.Order;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;
import java.util.List;
import java.util.Optional;

@Component
@Order(0) // run very early
public class IpAllowlistFilter extends OncePerRequestFilter {

    private final IpAllowlistProperties props;

    public IpAllowlistFilter(IpAllowlistProperties props) {
        this.props = props;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
        FilterChain filterChain) throws ServletException, IOException {
        if (!props.isEnabled()) {
            filterChain.doFilter(request, response);

```

```

        return;
    }
    String ip = Optional.ofNullable(request.getHeader("X-Forwarded-For"))
        .map(v -> v.split(",")[0].trim())
        .orElse(request.getRemoteAddr());
    List<String> allowed = props.parsed();
    if (!allowed.contains(ip)) {
        response.setStatus(HttpStatus.FORBIDDEN.value());
        response.getWriter().write("Forbidden (IP not allowed)");
        return;
    }
    filterChain.doFilter(request, response);
}
}

```

src/main/java/com/example/secureapi/security/SecurityConfig.java

java

```

package com.example.secureapi.security;

import com.example.secureapi.rate.RateLimitFilter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.WebSecurityCustomizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.csrf.CookieCsrfTokenRepository;
import org.springframework.security.web.firewall.HttpFirewall;
import org.springframework.security.web.firewall.StrictHttpFirewall;
import org.springframework.security.web.header.writers.ReferrerPolicyHeaderWriter;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import java.util.Set;

@Configuration
public class SecurityConfig {

    private final RateLimitFilter rateLimitFilter;
    private final IpAllowlistFilter ipAllowlistFilter;

```

```

public SecurityConfig(RateLimitFilter rateLimitFilter, IpAllowlistFilter ipAllowlistFilter) {
    this.rateLimitFilter = rateLimitFilter;
    this.ipAllowlistFilter = ipAllowlistFilter;
}

```

@Bean

```

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .cors(Customizer.withDefaults())
        .csrf(csrf -> csrf
            .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
            .ignoringRequestMatchers("/h2-console/**") // allow H2 console
        )
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/h2-console/**", "/actuator/health", "/api/csrf").permitAll()
            .anyRequest().permitAll()
        )
        .headers(headers -> headers
            .frameOptions(f -> f.sameOrigin()) // H2 console
            .referrerPolicy(r -> r.policy(ReferrerPolicyHeaderWriter.ReferrerPolicy.NO_REFERRER))
            .contentSecurityPolicy(csp -> csp
                .policyDirectives("default-src 'none'; " +
                    "img-src 'self'; " +
                    "script-src 'self'; " +
                    "style-src 'self'; " +
                    "connect-src 'self' http://localhost:5173; " +
                    "frame-ancestors 'none';"))
            );

    // Order: IP allowlist -> rate limit -> rest
    http.addFilterBefore(ipAllowlistFilter, UsernamePasswordAuthenticationFilter.class);
    http.addFilterBefore(rateLimitFilter, UsernamePasswordAuthenticationFilter.class);

    return http.build();
}

```

// Strict application-layer firewall (blocks encoded attacks / bad methods)

@Bean

```

public HttpFirewall httpFirewall() {
    StrictHttpFirewall fw = new StrictHttpFirewall();
    fw.setAllowUrlEncodedDoubleSlash(false);
}

```

```

fw.setAllowUrlEncodedPercent(false);
fw.setAllowedHttpMethods(Set.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));
return fw;
}

@Bean
public WebSecurityCustomizer webSecurityCustomizer(HttpFirewall firewall) {
    return web -> web.httpFirewall(firewall);
}
}

```

6) Run the API

```

bash

cd secure-api
mvn spring-boot:run

```

- API: `http://localhost:8080`
- H2 console: `http://localhost:8080/h2-console`

Part B — React Frontend (Vite)

1) Create project

```

bash

npm create vite@latest secure-ui -- --template react
cd secure-ui
npm i

```

2) Enable cross-site cookies (dev): Vite dev server is fine by default.

Create the following files/edits.

`index.html` (default from Vite is fine)

`src/api.js`

javascript

```
const API_BASE = 'http://localhost:8080';

function readCookie(name) {
  return document.cookie
    .split('; ')
    .find(row => row.startsWith(name + '='))
    ?.split('=')[1];
}

let csrfToken = null;

export async function ensureCsrf() {
  if (csrfToken) return csrfToken;
  const res = await fetch(`${API_BASE}/api/csrf`, {
    method: 'GET',
    credentials: 'include',
  });
  const data = await res.json();
  // Cookie "XSRF-TOKEN" also set by server; keep a copy in memory
  csrfToken = data.token || readCookie('XSRF-TOKEN');
  return csrfToken;
}

export async function getItem(q = '') {
  const params = q ? `?q=${encodeURIComponent(q)}` : '';
  const res = await fetch(`${API_BASE}/api/items${params}`, {
    credentials: 'include',
  });
  if (!res.ok) throw new Error(`GET /items failed: ${res.status}`);
  return await res.json();
}

export async function createItem(payload) {
  const token = await ensureCsrf();
  const res = await fetch(`${API_BASE}/api/items`, {
    method: 'POST',
    credentials: 'include', // include cookies so CSRF cookie is present
    headers: {
      'Content-Type': 'application/json',
      'X-XSRF-TOKEN': token,
    },
  },
```



```

    body: JSON.stringify(payload),
  });
  if (res.status === 429) throw new Error('Rate limited (429)');
  if (!res.ok) {
    const text = await res.text();
    throw new Error(`POST /items failed: ${res.status} ${text}`);
  }
  return await res.json();
}

```

src/ItemForm.jsx

jsx

```

import { useState } from 'react';
import { createItem } from './api';

export default function ItemForm({ onCreate }) {
  const [name, setName] = useState("");
  const [desc, setDesc] = useState("");
  const [busy, setBusy] = useState(false);
  const [error, setError] = useState(null);

  const submit = async (e) => {
    e.preventDefault();
    setError(null);
    if (!name.trim()) { setError('Name is required'); return; }
    if (desc.length > 1000) { setError('Description too long (max 1000)'); return; }
    setBusy(true);
    try {
      const item = await createItem({ name, description: desc });
      setName(""); setDesc("");
      onCreate?.(item);
    } catch (err) {
      setError(err.message);
    } finally {
      setBusy(false);
    }
  };

  return (
    <form onSubmit={submit} style={{ display:'grid', gap: 8, maxWidth: 520 }}>

```

```

    <input
      placeholder="Name"
      value={name}
      onChange={e => setName(e.target.value)}
    />
    <textarea
      placeholder="Description (HTML will be sanitized server-side)"
      value={desc}
      onChange={e => setDesc(e.target.value)}
      rows={4}
    />
    <button disabled={busy} type="submit">
      {busy ? 'Saving...' : 'Add Item'}
    </button>
    {error && <div style={{ color:'red' }}>{error}</div>}
  </form>
);
}

```

src/ItemList.jsx

```

jsx

export default function ItemList({ items }) {
  if (!items?.length) return <div>No items yet.</div>;
  return (
    <ul style={{ padding: 0, listStyle: 'none', display:'grid', gap: 12 }}>
      {items.map(it => (
        <li key={it.id} style={{ border:'1px solid #ddd', padding:12, borderRadius:8 }}>
          <div><b>{it.name}</b></div>
          {/* React escapes by default. We do NOT dangerouslySetInnerHTML here. */}
          <div style={{ whiteSpace:'pre-wrap', opacity:.9 }}>{it.description}</div>
        </li>
      ))}
    </ul>
  );
}

```

src/App.jsx

```

jsx

```

```

import { useEffect, useState } from 'react';
import { ensureCsrf, getItems } from './api';
import ItemForm from './ItemForm';

export default function App() {
  const [items, setItems] = useState([]);
  const [q, setQ] = useState("");
  const [error, setError] = useState(null);
  const [loading, setLoading] = useState(true);

  const load = async (query = "") => {
    try {
      setLoading(true);
      // Ensure CSRF cookie exists for subsequent POSTs
      await ensureCsrf();
      const data = await getItems(query);
      setItems(data);
      setError(null);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  };

  useEffect(() => { load(); }, []);

  return (
    <div style={{ margin:'2rem auto', maxWidth:800, display:'grid', gap:16 }}>
      <h2>Secure Items</h2>

      <div style={{ display:'flex', gap:8 }}>
        <input
          placeholder="Search by name..."
          value={q}
          onChange={(e) => setQ(e.target.value)}
          onKeyDown={(e) => e.key === 'Enter' && load(q)}
          style={{ flex:1 }}
        />
        <button onClick={() => load(q)}>Search</button>
        <button onClick={() => { setQ(""); load(""); }}>Clear</button>
      </div>

```

```

    <ItemForm onCreate={({it} => setItems(prev => [it, ...prev]))} />

    {loading ? <div>Loading...</div> :
      error ? <div style={{ color:'red' }}>{error}</div> :
      <div><hr/><ItemList items={items} /></div>
    }
  </div>
);
}

```

src/main.jsx

```

jsx

import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App'

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)

```

Run the UI:

```

bash

npm run dev
# Open the printed URL (usually http://localhost:5173)

```

How each security feature is covered

- **CORS:** Only `http://localhost:5173` can call the API from browsers (`CorsConfig`, `application.yml`).
- **CSRF:** Enabled with `CookieCsrfTokenRepository`. UI calls `/api/csrf` once to receive the `XSRF-TOKEN` cookie and then sends `X-XSRF-TOKEN` header on POST.
- **Rate limiting:** `RateLimitFilter` uses `Bucket4j` (100 req/min per IP) → returns **429** when exceeded.

- **SQL injection:** Spring Data JPA uses **parameter binding**; no string concatenation in queries.
- **NoSQL injection:** `ItemDocService` uses `Criteria` + `Pattern.quote()` (safe regex) and avoids `$where`.
- **XSS:** Input sanitized server-side with **Jsoup Safelist** before save; React renders as text (no `dangerouslySetInnerHTML`). CSP headers also reduce risk.
- **Firewalls:** `StrictHttpFirewall` blocks suspicious encodings/methods at the app layer.
- **VPN/IP allowlisting:** Turn on allowlist by setting

yaml

```
app:
  ip-allowlist:
    enabled: true
    addresses: 127.0.0.1,0:0:0:0:0:0:1
```

(For production, put the API behind a VPN/VPC and/or enable **mTLS** via `server.ssl.client-auth=need` with keystores.)

Quick test

1. Start API (`mvn spring-boot:run`) and UI (`npm run dev`).
2. In UI, add an item with description containing `<script>alert(1)</script>` → saved value is sanitized, alert does **not** run.
3. Try spamming Add many times → eventually **429 Too Many Requests**.
4. If you enable IP allowlist and your IP isn't in list → you'll get **403 Forbidden**.

If you want this adjusted for **JWT stateless auth**, I can swap CSRF for JWT validation and keep all the other protections intact.

why hardcoding `http://localhost:5173`, can you give array of allowed url's in `applications.yml`

Good question 👍 — hardcoding a single origin (like `http://localhost:5173`) isn't flexible. Instead, we can maintain an **array of allowed URLs** in `application.yml` and load them dynamically into the CORS config.

Here's how you can do it **step by step**:

1. Update `application.yml`

```
yml

server:
  port: 8080

app:
  cors:
    allowed-origins:
      - "http://localhost:5173"
      - "http://localhost:3000"
      - "https://my-frontend.example.com"
```

2. Create a `CorsProperties` class

```
java

package com.example.demo.config;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;

import java.util.List;

@Configuration
@ConfigurationProperties(prefix = "app.cors")
public class CorsProperties {
    private List<String> allowedOrigins;


    public List<String> getAllowedOrigins() {
        return allowedOrigins;
    }

    public void setAllowedOrigins(List<String> allowedOrigins) {
        this.allowedOrigins = allowedOrigins;
    }
}
```

```
}  
}
```

3. Update the CORS configuration

java

```
package com.example.demo.config;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.web.cors.CorsConfiguration;  
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;  
import org.springframework.filter.CorsFilter;  
  
@Configuration  
public class WebConfig {  
  
    private final CorsProperties corsProperties;  
  
    public WebConfig(CorsProperties corsProperties) {  
        this.corsProperties = corsProperties;  
    }  
  
    @Bean  
    public CorsFilter corsFilter() {  
        CorsConfiguration config = new CorsConfiguration();  
        config.setAllowCredentials(true);  
        config.setAllowedOrigins(corsProperties.getAllowedOrigins()); //  load from  
application.yml  
        config.addAllowedHeader("*");  
        config.addAllowedMethod("*");  
  
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();  
        source.registerCorsConfiguration("/*", config);  
        return new CorsFilter(source);  
    }  
}
```

✓ Now CORS is flexible

- You can add or remove allowed origins in `application.yml` without touching Java code.
 - Example:
 - Local dev → `http://localhost:5173`
 - Another dev frontend → `http://localhost:3000`
 - Production → `https://my-frontend.example.com`
-

👉 Do you also want me to update the **React frontend** so it automatically picks the correct API base URL depending on whether it's running in **dev** or **prod**?

yes

Perfect 👍 let's make your **React frontend** automatically pick the correct API base URL depending on environment (dev/prod).

We'll do it step by step from scratch:

1. Create a React project

If you don't already have one:

```
bash

npx create-react-app frontend
cd frontend
```

(or if you prefer Vite)

```
bash

npm create vite@latest frontend --template react
cd frontend
npm install
```

2. Setup environment files

Inside `frontend/`, create:

`.env.development`

env

`REACT_APP_API_BASE_URL=http://localhost:8080`

`.env.production`

env

`REACT_APP_API_BASE_URL=https://my-backend.example.com`

3. Create an API helper

👉 `src/api.js`

javascript

```
const API_BASE_URL = process.env.REACT_APP_API_BASE_URL;

export async function fetchHello() {
  const response = await fetch(`${API_BASE_URL}/api/hello`);
  if (!response.ok) {
    throw new Error("Network response was not ok");
  }
  return await response.text();
}
```

4. Use it in your React component

👉 `src/App.js`

javascript

```
import React, { useEffect, useState } from "react";
import { fetchHello } from "./api";

function App() {
```

```
const [message, setMessage] = useState("");

useEffect(() => {
  fetchHello()
    .then((msg) => setMessage(msg))
    .catch((err) => setMessage("Error: " + err.message));
}, []);

return (
  <div>
    <h1>React + Spring Boot Demo</h1>
    <p>Backend says: {message}</p>
  </div>
);
}

export default App;
```

5. Run the frontend

For dev:

```
bash

npm start
```

For prod (build + serve):

```
bash

npm run build
serve -s build
```

✓ Now:

- In dev → API calls `http://localhost:8080/api/...`
 - In prod → API calls `https://my-backend.example.com/api/...`
-

⚡ Bonus: If you want multiple frontend URLs (like we did in Spring Boot `application.yml`), React doesn't need to care — it just uses `REACT_APP_API_BASE_URL`, and backend CORS config decides whether to allow it.

Do you also want me to **add CSRF, XSS protection & SQL injection handling** in the Spring Boot API code with examples next, so frontend is fully secure?

mongodb give docker run command to configure

Got it 🚀 — you want to spin up **MongoDB in Docker** from scratch. Here's the step-by-step with the right `docker run` command:

1. Run MongoDB container

```
bash

docker run -d \
  --name mongodb \
  -p 27017:27017 \
  -e MONGO_INITDB_ROOT_USERNAME=admin \
  -e MONGO_INITDB_ROOT_PASSWORD=secret \
  mongo:6.0
```

Explanation:

- `-d` → detached mode (runs in background)
 - `--name mongodb` → container name
 - `-p 27017:27017` → expose MongoDB on host port `27017`
 - `-e MONGO_INITDB_ROOT_USERNAME=admin` → root username
 - `-e MONGO_INITDB_ROOT_PASSWORD=secret` → root password
 - `mongo:6.0` → official MongoDB image (v6.0)
-

2. Verify container is running

```
bash
```

```
docker ps
```

You should see `mongo:6.0` .

3. Connect to Mongo shell inside container

```
bash
```

```
docker exec -it mongodb mongosh -u admin -p secret
```

Now you're inside the MongoDB shell 🎉.

4. (Optional) Create a new database & user

Inside `mongosh` :

```
javascript
```

```
use mydb;
```

```
db.createUser({  
  user: "myuser",  
  pwd: "mypassword",  
  roles: [{ role: "readWrite", db: "mydb" }]  
});
```

Now you have a **non-root user** with access only to `mydb` .

5. Connection string for Spring Boot

Your `application.yml` :

```
yaml
```

```
spring:  
  data:  
    mongodb:  
      uri: mongodb://myuser:mypassword@localhost:27017/mydb
```

⚡ If you also want **Mongo Express (web UI)**, run:

```
bash

docker run -d \
  --name mongo-express \
  -p 8081:8081 \
  -e ME_CONFIG_MONGODB_ADMINUSERNAME=admin \
  -e ME_CONFIG_MONGODB_ADMINPASSWORD=secret \
  -e ME_CONFIG_MONGODB_SERVER=mongodb \
  --link mongodb:mongo \
  mongo-express
```

Then open 🖱️ <http://localhost:8081>.

Do you want me to also **add MongoDB integration in your Spring Boot API** (with secure CRUD endpoints and XSS/SQL injection safe practices)?

mongodb vs mongo-express?

Great question 🖱️ — let's break it down:

● MongoDB

- **What it is:**

MongoDB itself is the **database server** (NoSQL database).

It stores your collections, documents, indexes, etc.

- **How you use it:**

- Your **Spring Boot API** or any backend connects directly to MongoDB using a driver.
- You run queries with `mongosh` CLI or via your application.

- **Docker run example:**

```
bash

docker run -d \
  --name mongodb \
```

```
-p 27017:27017 \  
-e MONGO_INITDB_ROOT_USERNAME=admin \  
-e MONGO_INITDB_ROOT_PASSWORD=secret \  
mongo:6.0
```

- **Purpose:** The actual **engine** that stores and retrieves your data.

● Mongo Express

- **What it is:**

Mongo Express is a **web-based admin UI** for MongoDB.

Think of it like **phpMyAdmin** for MySQL or **pgAdmin** for PostgreSQL.

- **How you use it:**

- You can **view databases, collections, documents** in the browser.
- You can **insert, edit, delete data** manually through a web UI.
- Useful for **debugging** and **managing MongoDB without CLI**.

- **Docker run example:**

```
bash  
  
docker run -d \  
  --name mongo-express \  
  -p 8081:8081 \  
  -e ME_CONFIG_MONGODB_ADMINUSERNAME=admin \  
  -e ME_CONFIG_MONGODB_ADMINPASSWORD=secret \  
  -e ME_CONFIG_MONGODB_SERVER=mongodb \  
  --link mongodb:mongo \  
  mongo-express
```

- **Access:**

Open <http://localhost:8081> to see UI.

✓ When to use what?

- Use **MongoDB** → always (it's the database backend).
- Use **Mongo Express** → optional, only if you want a **quick admin panel** to see/manage data easily.

In production:

- You'd normally **not run mongo-express** (security risks if exposed).

- Instead, you'd use professional tools like **MongoDB Compass** (desktop GUI) or MongoDB Atlas web UI if you host in the cloud.
-

👉 In your case (Spring Boot + React project),

- MongoDB = required (backend storage).
 - Mongo Express = optional (for quick debugging/testing).
-

Do you want me to show you **how to add both MongoDB and Mongo Express in a single** `docker-compose.yml` so you can spin them up together?