

## **Web Engineering**

# **Security**

Adrian Herzog

*(basierend auf der Arbeit von Michael Faes, Michael Heinrichs & Prof. Dierk König)*

# **Security-Grundlagen**

# Authentifizierung & Autorisierung

**Authentifizierung:** «*Wer bist du?*»

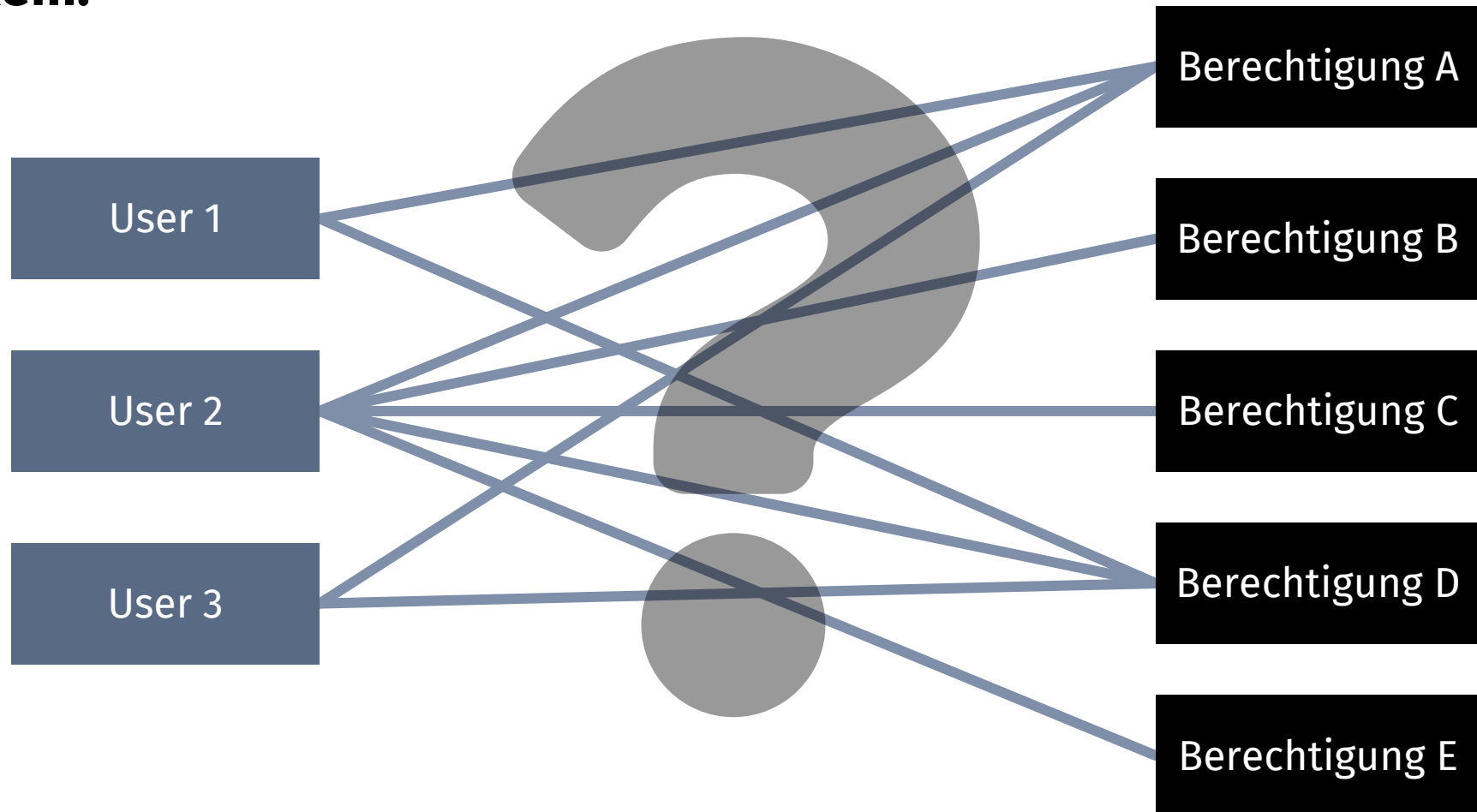
Prozess, der Identität eines Benutzers *verifiziert*, z. B. durch Abfragen von Benutzername & Passwort

**Autorisierung:** «*Welche Operationen darfst du ausführen?*»

Prozess, der überprüft, welche *Berechtigungen* ein (authentifizierter) Benutzer hat. (Auch *Access Control*)

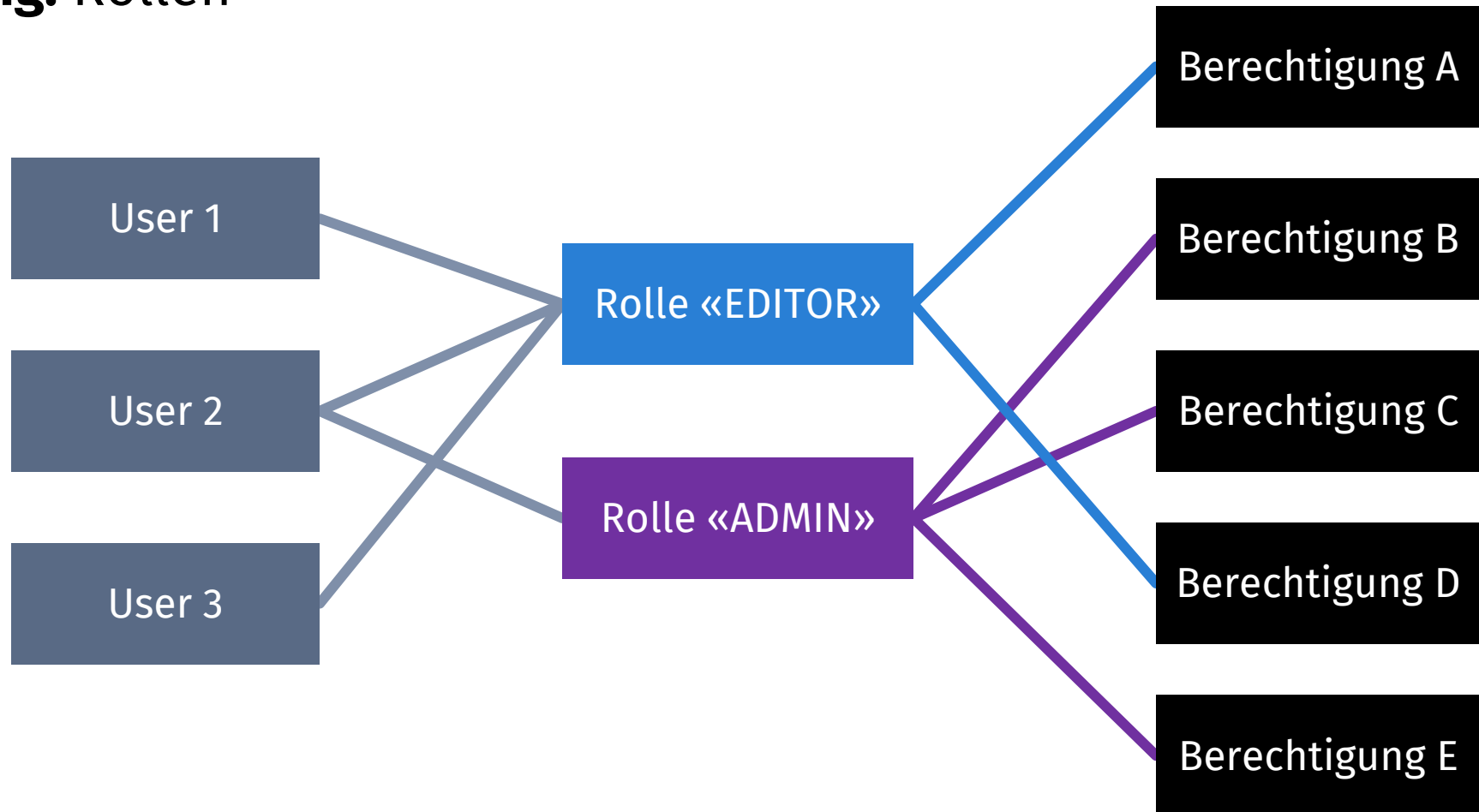
# Berechtigungen verwalten

**Problem:**



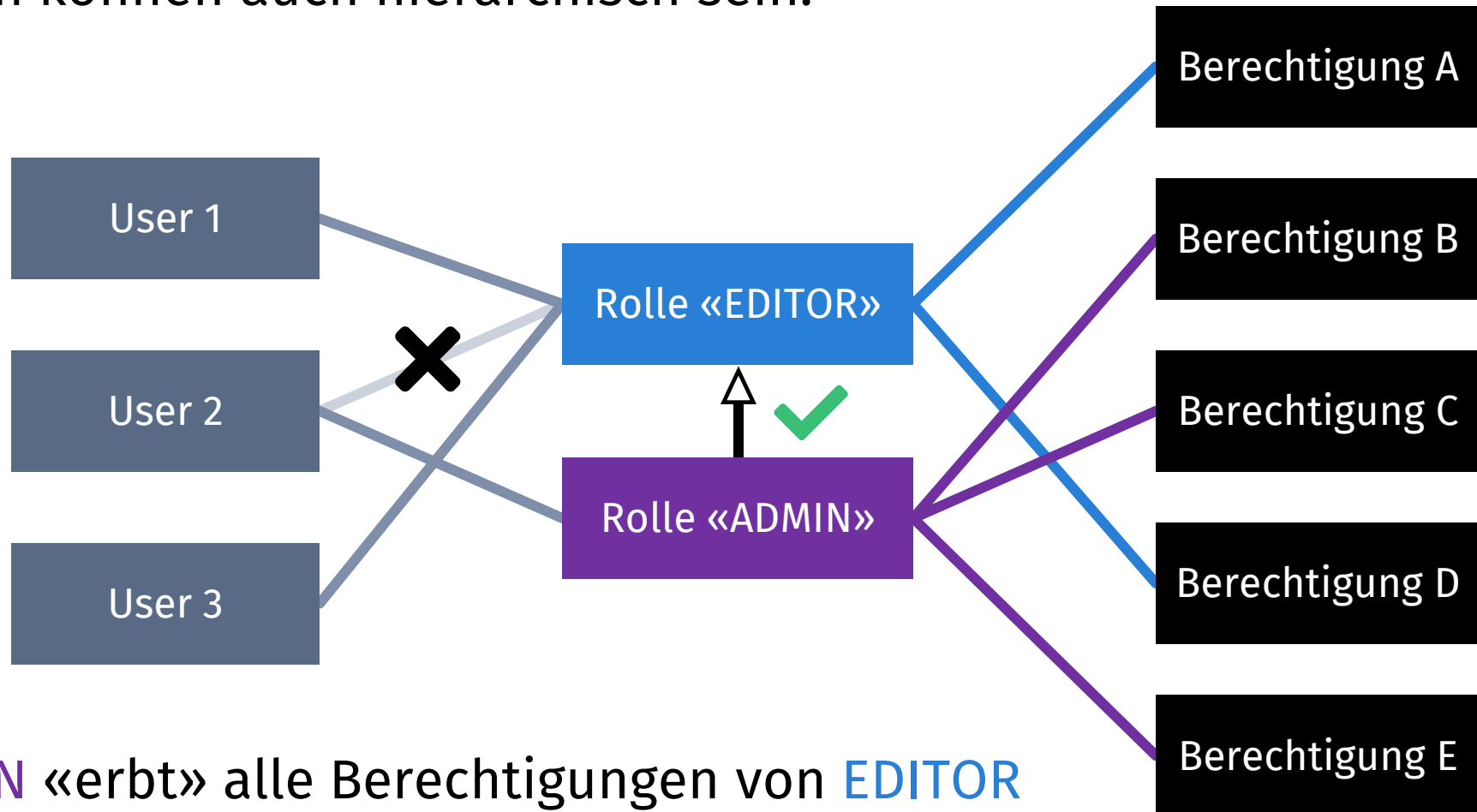
# Role Based Access Control (RBAC)

## Lösung: Rollen



# Hierarchische Rollen

Rollen können auch hierarchisch sein:



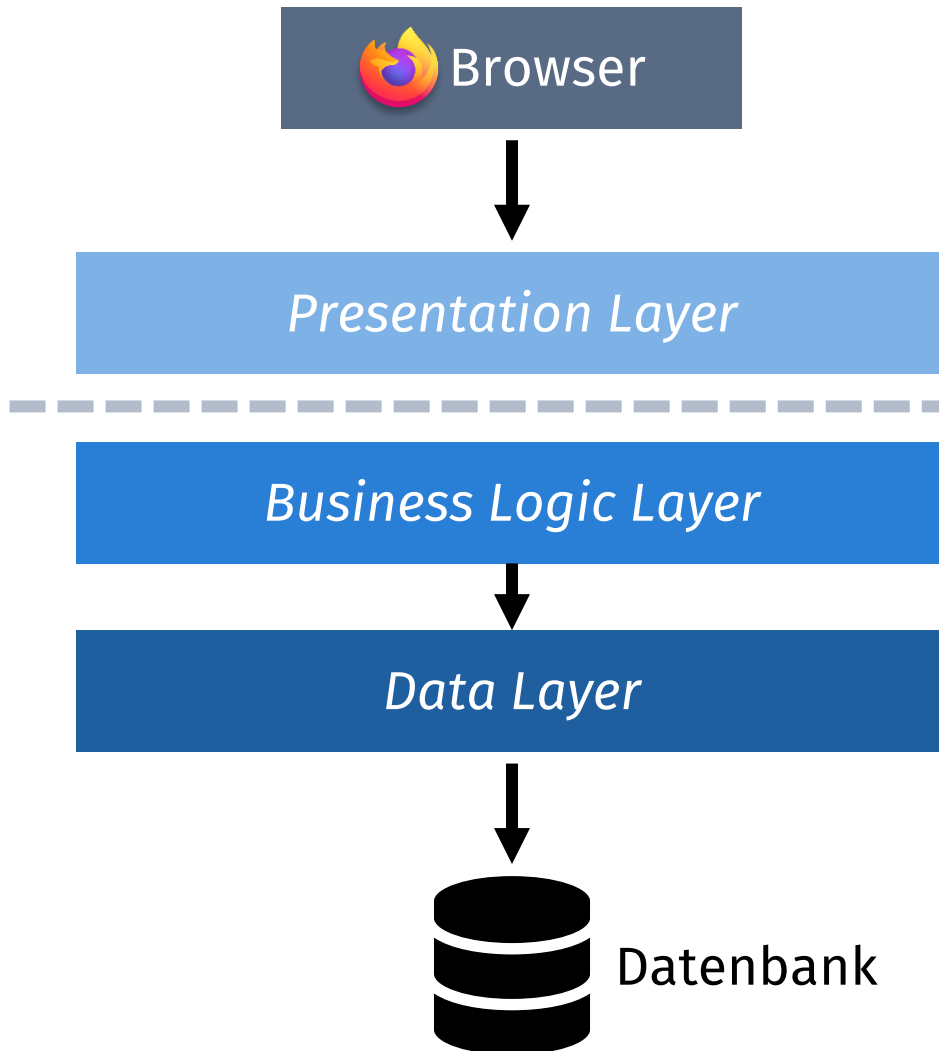
**ADMIN** «erbt» alle Berechtigungen von **EDITOR**

# Whitelisting vs. Blacklisting

(Auch «*allow lists*» und «*deny lists*» / «*block lists*»)

Art	Default	Konfiguration	Konsequenzen
<b>Whitelist</b>	Alles verboten	Selektiv erlauben	Vergessene Dinge funktionieren nicht
<b>Blacklist</b>	Alles erlaubt	Selektiv verbieten	Vergessene Dinge sind ungeschützt
<b>Mix</b>	Whitelist oder Blacklist, je nach Berechtigungsart	Selektiv erlauben oder verbieten	Flexibel, aber fehleranfällig

# Path vs. Resource Protection



## Path Protection:

Berechtigungen basierend auf Pfad, z. B. `example.com/contacts`

- Viel Hilfe von Tools

## Resource Protection:

Berechtigung basierend auf irgendwelchen «Ressourcen»

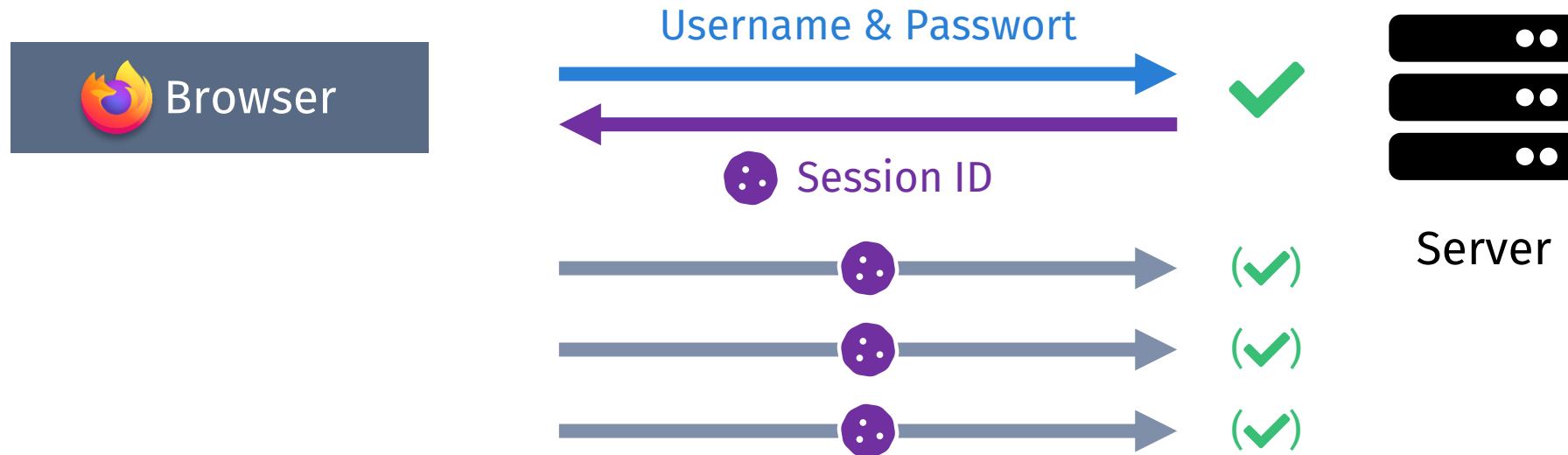
- Z. B. für bestimmte Datensätze
- Muss meistens selber implementiert werden...



# Security mit HTTP

HTTP ist zustandloses Protokoll! Bei jeder Anfrage authentifizieren?

**Im Prinzip ja.** Aber nur einmal *mit Passwort*:

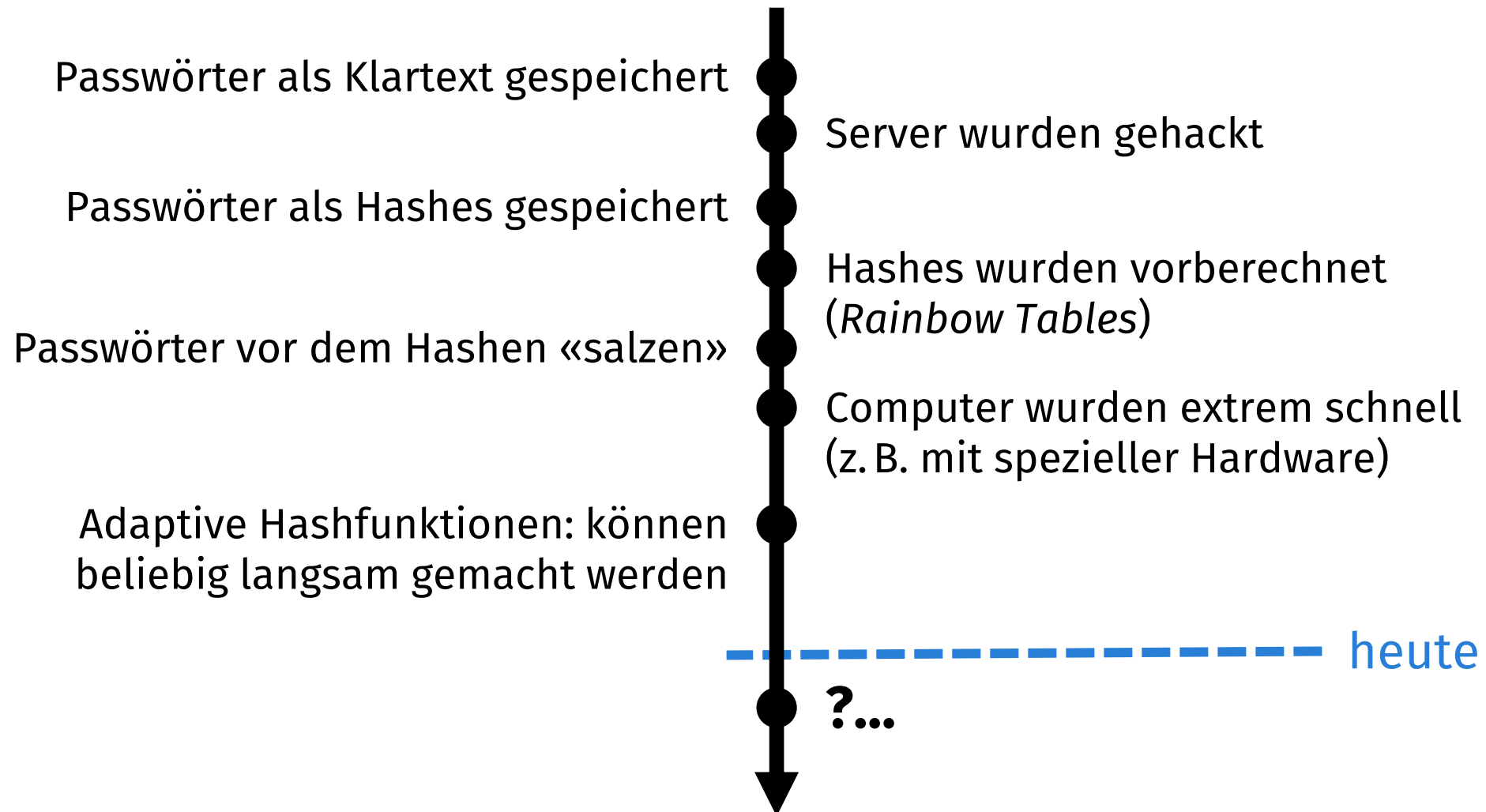


Session ID wird typischerweise als «Cookie» mitgeschickt.

Muss geheim bleiben! → **HTTPS**

# Passwörter speichern?

Eine kurze Geschichte vom Speichern von Passwörtern...



Regel 1:

**Security-Code nie selber schreiben**

# Spring Security

# Spring Security

## Quickstart mit Spring Boot:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

## Default-Konfiguration:

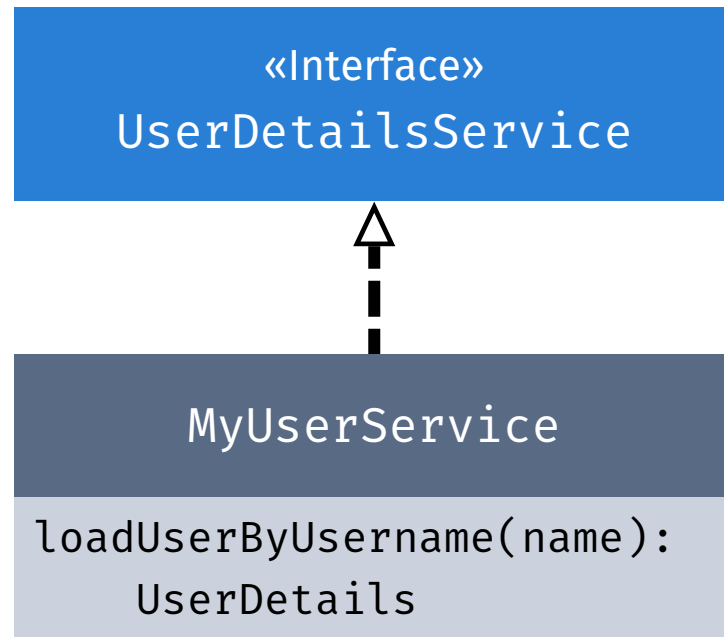
- Erstellt Default-User mit zufällig generiertem Passwort (Konsole)
- Speichert Passwort (gesalzen und gehasht) in Hauptspeicher
- Verlangt authentifizierten User für *jede* Anfrage (alle Pfade)
- Erstellt Login-Formular und Logout-Endpoint
- Erstellt und verwaltet Sessions via Cookies
- Schützt vor CSRF



# Security-Konfiguration

Zwei zentrale Komponenten, um Spring Security zu konfigurieren:

## Authentifizierung



## Autorisierung



# Autorisierung in Spring Security

Autorisierungs-Konfig durch ein Bean vom Typ `SecurityFilterChain`

```
@Configuration   
public class WebSecurityConfig {  
    @Bean  
    public SecurityFilterChain filterChain(HttpSecurity http) ... {  
        return http  
            .authorizeHttpRequests(req -> req  
                .requestMatchers("/").permitAll()  
                .requestMatchers("/user/**").authenticated()  
                .anyRequest().hasRole("ADMIN"))  
            .formLogin(login -> login  
                .loginPage("/login").permitAll())  
            .build();  
    }  
}
```

öffentlich

eingeloggt


mit Rolle

eigener Controller  
für Login-Formular

`requestMatchers()` erlaubt Muster mit `*` (ohne `'/'`) oder `**` (inkl. `'/'`)

# Übung 1: Hello, Spring Security!

- a) Pule die neue Vorlage. Füge Spring Security als Dependency hinzu und prüfe, dass die App jetzt geschützt ist. Beachte das Passwort, das auf der Konsole ausgegeben wird (Benutzername: «user»).
- b) Konfiguriere Spring Security so, dass die Startseite und die Kategorien-Seite öffentlich zugänglich sind, aber die Seiten der einzelnen Wunschlisten nur authentifizierten Users angezeigt werden. Denke auch an die statischen Inhalte (CSS, Bilder).



Beachte auch  
die Hinweise im  
README.md



# Feinere Kontrolle über Autorisierung

Elemente selektiv anzeigen, basierend auf Benutzer-Information:

## Option 1: In Pebble-Template

```
{% if request.userPrincipal != null %}
```

wird nur eingeloggten  
Users angezeigt

```
{% if request.isUserInRole("ADMIN") %}  
<div>  
    <a href="/edit-list/3">Liste bearbeiten</a>  
</div>  
{% endif %}
```

wird nur ADMINS  
angezeigt

**Vorsicht:** Bietet keine echte Security, «nur» gute Benutzer-Experience!

- URL </edit-list/3> muss zusätzlich Server-seitig geschützt werden, ansonsten: einfach von Hand </edit-list/3> in Browser eingeben...

## Option 2: In Controller:

org.springframework  
security.core

```
@GetMapping("/edit-list/{id}")  
public String editList(..., Authentication auth) {  
    ... auth.getAuthorities() ...  
}
```

`getAuthorities()` gibt eine Sammlung von `GrantedAuthoritys` zurück:

```
public interface GrantedAuthority {  
    String getAuthority();  
}
```

z. B. `ROLE_ADMIN`

!

Wichtig bei Autorisierung und bei Konfig von Authentifizierung:  
`getAuthority()` gibt Rollen mit «ROLE\_»-Präfix zurück.

# Übung 2: Templates & Security

- a) Ändere die Pebble-Templates so ab, dass nur eingeloggte User die Links zu den Wunschlisten sehen. Anonyme User sehen die Wunschlisten ohne Links.
- b) Verstecke auch die Formulare zum Erstellen/Löschen von Dingen vor anonymen Usern.

# Authentifizierung mit Spring Security

Authentifizierungs-Konfig durch Bean vom Typ `UserDetailsService`:

```
public interface UserDetailsService {  
    UserDetails loadUserByUsername(String username);  
}
```

Einfache, *nicht persistente* Konfiguration:

@Bean



```
public UserDetailsService userDetailsService() {  
    var builder = User.withDefaultPasswordEncoder();  
    var admin = builder  
        .username("admin").password("admin")  
        .roles("ADMIN").build();  
    return new InMemoryUserDetailsManager(admin);  
}
```



## Persistente User: Eigene Implementation von UserDetailsService und UserDetails, z.B. durch JPA/Spring Data:

```
@Service
public class UserService implements UserDetailsService {

    private final UserRepository userRepo;
    ...
    // eigenes Repository

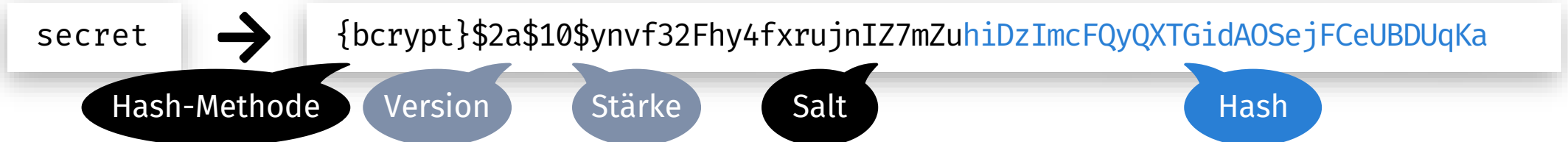
    public UserDetails loadUserByUsername(String username) {
        // use userRepo to retrieve user from DB
        // if not found, throw UsernameNotFoundException
    }
}
```

```
@Entity
public class User implements UserDetails {
    // need to implement UserDetails methods...
}
```

# Speichern von Passwörtern

Bevor Passwort in DB gespeichert wird, muss es gehasht werden:

```
public void createUser(String username, String password,
                      Set<String> roles) {
    var encoder = PasswordEncoderFactories
        .createDelegatingPasswordEncoder();
    var encoded = encoder.encode(password);
    var user = new User(username, encoded, roles);
    userRepo.save(user);
}
```

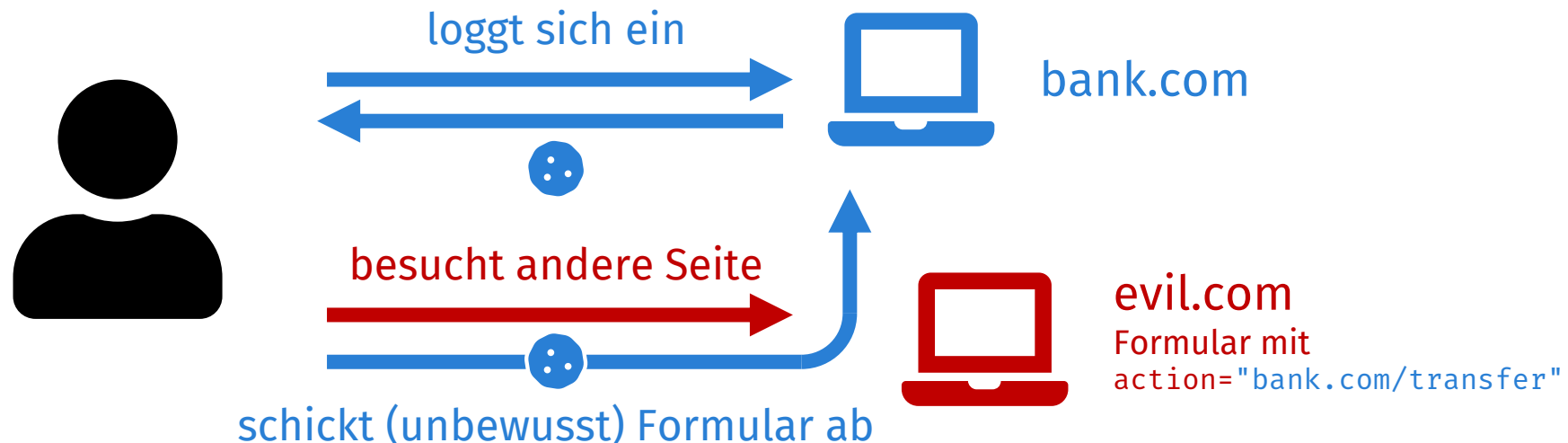


Spring Security wählt «geeignete» Hash-Methode und Stärke.

# Problem mit Cookies: CSRF

**Risiko von Cookies:** Browser schickt Cookies von eingeloggten Usern *immer* an Server mit – Ort von Link oder Formular spielt keine Rolle.

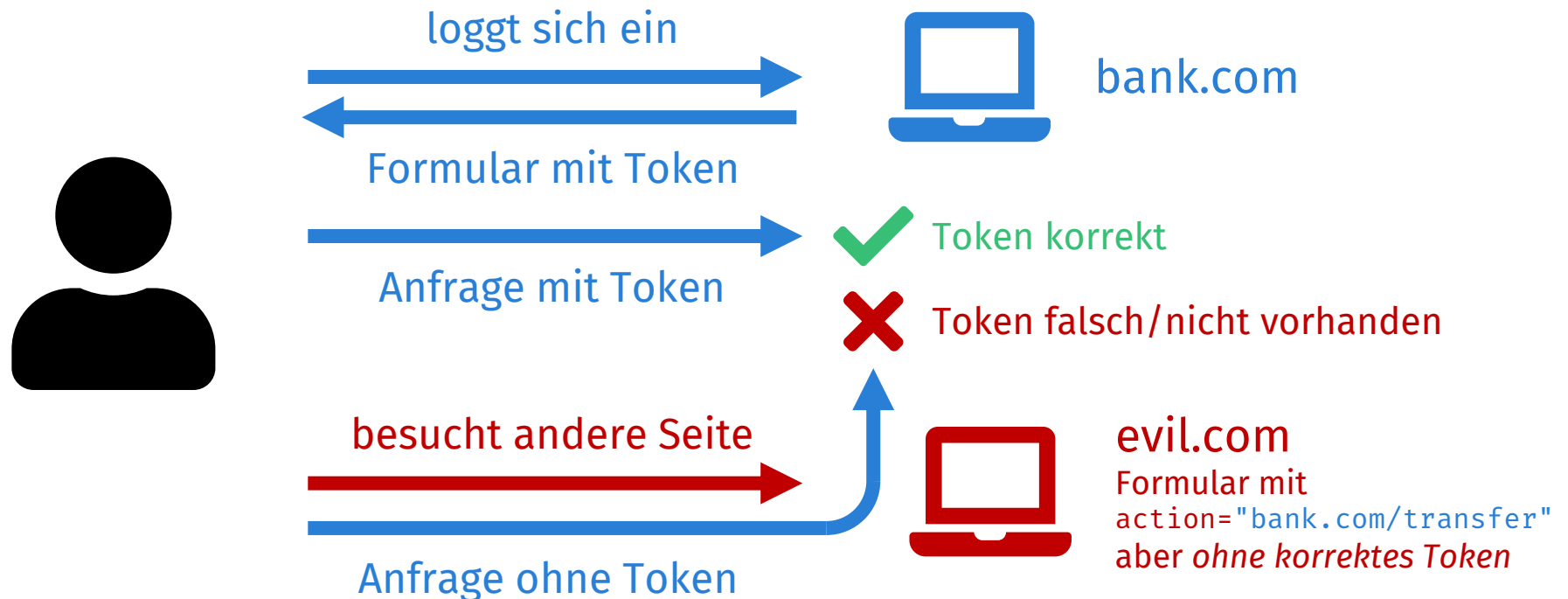
**Cross site request forgery (CSRF):** Bösertige Website **evil.com** präpariert Formular für **bank.com** und schickt es für Besucher ab – falls Besucher gerade bei **bank.com** eingeloggt ist, wird sein Cookie mitgeschickt!



# Schutz vor CSRF

Um vor CSRF zu schützen, darf Server nur POST-Requests akzeptieren, die von «eigenen» Formularen stammen.

**Lösung:** In *jedes Formular* ein zufällig generiertes Token einfügen und bei jeder POST-Anfrage prüfen, dass gleiches Token enthalten ist.





# CSRF mit Spring Security

Mit Default-Konfiguration akzeptiert Spring Security keine Formulare ohne CSRF-Token. 👍

**Bedeutet:** Müssen in *jedes* Formular verstecktes Feld einfügen:

```
<input type="hidden"  
      name="{{ request.getAttribute("_csrf").parameterName }}"  
      value="{{ request.getAttribute("_csrf").token }}">
```

**Bedeutet auch:** H2-Console funktioniert nicht mehr...

Alternative: Mit IntelliJ Ultimate direkt auf DB-Datei zugreifen.



Demo

# Fragen?

