Webentwicklung

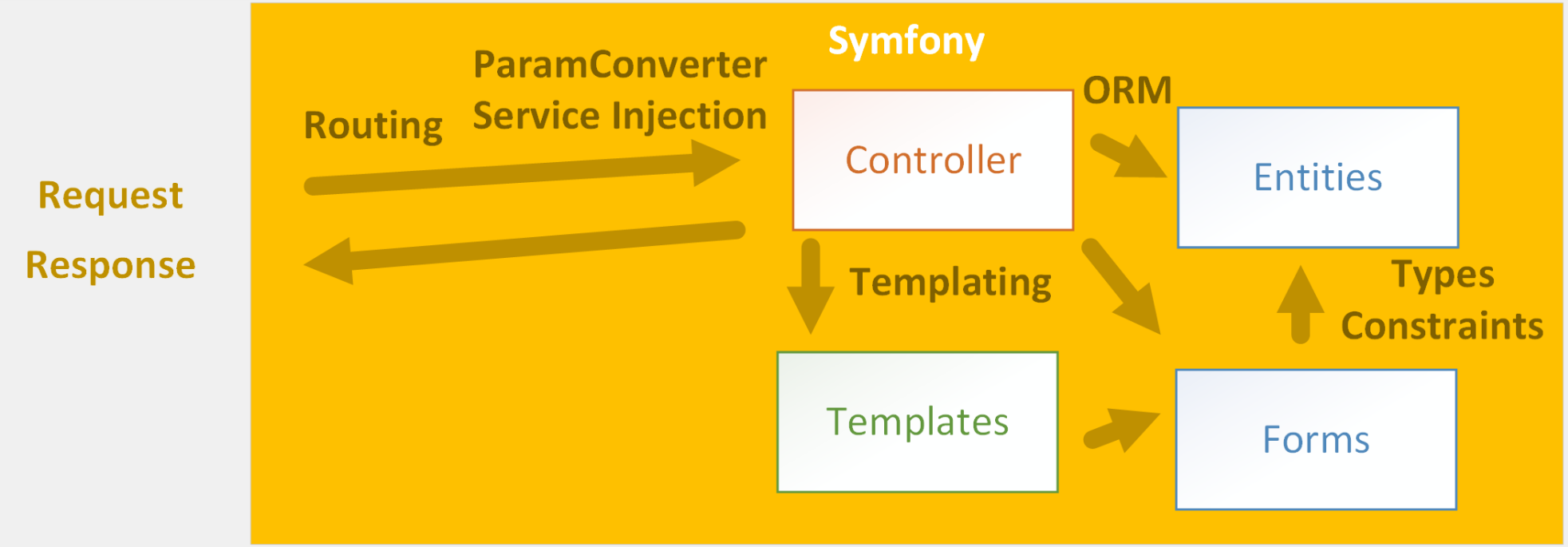
Backend: Symfony (Vertiefung)

Inhalt dieser Einheit

1. Verzeichnisstruktur
2. Sessions
3. Internationalisierung
4. Zugriffskontrolle
    - **Authentisierung**: Firewall, Provider, Hashing
    - **Autorisierung**: Rollenbasiert & Attributbasiert
5. Kommandozeile
6. Debugging

Symfony: Wiederholung

- Framework-Charakter:
  - definiert Arten von Elementen und deren Beziehung
  - lässt Lücken, müssen während der Entwicklung gefüllt werden



Symfony: Zentrale Ideen

- **Controller**: verarbeiten Requests, erzeugen Responses
  - werden über **Routen** angesprochen
    - mit Platzhaltern und Bedingungen
    - Extras wie ParamConverter für leichteren ORM-Zugriff
  - Bindeglied zwischen Model (**Entities**) und View (**Templates**)
- **Formulare**: echte Objekte, für Markup & Submit
  - erstellen mit FormBuilder
    - div. Feld-Typen & evtl. Start-Daten (für Markup-Erzeugung)
    - div. Constraints (für Submit-Behandlung)
  - Rendern: Themes und Template-Helfer
  - Formulare für Entities: autom. Feld-Typen und Constraints
- **Philosophie**:
  - *Konvention über Konfiguration*
  - *Modellierung in Model-Klassen*

Symfony: konkretes Beispiel

- Supero-Website als Symfony-Anwendung
  - https://github.com/fzieris/php-demo-supero-symfony
- README.md
  - erklärt, wie man die Supero-Webseite lokal und auf Heroku installiert
- development.md
  - erläutert die Entwicklungsschritte von der einfachen PHP-Variante hin zur Symfony-Anwendung
- evtl. für eine gewisse Übungsaufgabe hilfreich 🤔

Heute: ein etwas tieferer

Einblick in **Symfony**

Symfony-Projekt-Struktur

Symfony-Verzeichnisstruktur

- in jedem Projekt: immer gleicher Aufbau
  - `bin/`: Ausführbares (`php bin/console`)
  - `config/`: Konfiguration (überall gleich)
  - **`public/`**: nach außen sichtbar, **für Webserver**
  - **`src/`**: PHP-Klassen, **eigtl. Anwendung**
  - **`templates/`**: die **View** in “MVC”
  - `var/`: zur Laufzeit erzeugt (Cache, Logs)
  - `vendor/`: wiederverwendete Komponenten
  - `.env`: Konfiguration (aktuelle Maschine)
  - `.env.dist`: Vorlage für .env-Datei
  - `composer.json`: Definition der Abhängigkeiten

Wdh.: Arten von Web-Ressourcen

- **statische Web-Ressourcen**:
  - HTML-Dokumente, CSS-Stylesheets, JavaScript-Dateien
  - Request-Ziel ist eine Datei im Dateisystem des Servers
  - Webserver liefert *Dateiinhalt* bei HTTP-Request aus
- **dynamische Web-Ressourcen**:
  - Request-Ziel ist ein Programm/-anweisung
  - Webserver startet Programm bei Request
  - Programmausgabe wird als Response gesendet
- *PHP-Skript?*
  - dynamische Ressource (falls vom Webserver angesprochen)
- *Twig-Template?*
  - Hilfsdatei für PHP-Skript, *gar keine* Web-Ressource

Symfony-Verzeichnis: `public/`

- `index.php`
  - automatisch von Symfony erzeugt
  - richtet *Auto-Loading* ein
  - ruft den Kernel von Symfony auf:
    - leitet Request an Kernel weiter
    - sendet Response vom Kernel an Client
  - das war's!
- sonst: nur *statische Ressourcen*
  - CSS-Stylesheets, JavaScript-Dateien, Bilder
- alles Weitere ist von außen *nicht* erreichbar
  - (Apache: per `DocumentRoot`-Direktive; Heroku: in `Procfile`)
  - Datenbank-Konfiguration, Logfiles, …

Model-View-Controller?

- 90% der Entwicklungszeit: in `src/` und `templates/`

```
templates/        # View
src/
  Controller/     # Controller
  Entity/         # Model
```

- weitere `src/`-Ordner je nach installierten Komponenten:

```
  Command/        # Kommandozeilen-Befehle
  DataFixtures/   # Standard- oder Test-Daten
  Form/           # Formulare
  Migrations/     # bei mehreren DB-Schema-Versionen
  Repository/     # für häufig gebrauchte Queries
  Security/       # für Zugangsbeschränking
  Twig/           # Twig-Ergänzungen
  ...
```

- Symfony erkennt & lädt Klassen automatisch

Konfiguration

- Zwei Arten von Konfiguration:
  1. Allgemeine Konfiguration: `config/`
  2. Infrastruktur, auf jeder Machine anders: `.env`-Datei
     - Vorlage: `.env.dist`-Datei
     - Definiert u.a. die die Datenbank-Verbindung
- Grundidee *Environments*:
  - Anwendung hat verschiedene Betriebsmodi
    - mindestens `'dev'` und `'prod'` (evtl. auch `'test'`)
  - Welcher Modus: steht in `.env`-Datei
  - Unterschiede:
    - Welche Komponenten sind aktiv? (siehe `config/bundles.php`)
    - Wie viel Caching? (`'prod'`: viel)
    - Wie viel Logging? (`'dev'`: viel)

Quelle: https://symfony.com/doc/current/configuration.html

Einige wichtige Konfig-Dateien

- Alle diese Dateien haben “sinnvolle” Default-Inhalte
  - `config/routes.yaml`: Routen-Definitionen
    - (Alternative zu den Annotationen direkt an den Controllern)
  - `config/packages/framework.yaml`: Symfony an sich
  - `config/packages/security.yaml`: Zugriffskontrolle
  - `config/packages/translation.yaml`: Mehrsprachigkeit
  - `config/packages/twig.yaml`: Verhalten der Template-Engine
- Mehr zu Konfiguration von Symfony-Anwendungen:
  - https://symfony.com/doc/current/configuration.html

Sessions

Sessions: Was und Wofür?

- Beispiele:
  - Warenkorb beim Online-Shopping
  - Zugang zu geschützten Inhalten
    - inkl. Logout-Möglichkeit (nicht wie bei HTTP-Basic-Auth)
- Serverseitige Speicherung von nutzerbezogenen Daten
  - “Wiedererkennung” von Nutzern zwischen mehreren Requests

Sessions?

- HTTP ist zustandslos, es gibt kein Gedächtnis
  - jede Anfrage an den Server ist wie die erste Anfrage
- Trick: Cookies
  - Server setzt spezielles HTTP-Header-Feld in Response
    - Request: *“Ich hätte gerne einen Wikipedia-Artikel.”*
    - Response: *“Alles klar, X23njv3ga. Hier ist dein Artikel.”*
  - kompatibler Client schickt Wert bei jeder Folge-Anfrage mit
    - Request: *“Hier ist wieder X23njv3ga, ich hätte gerne noch einen Artikel.”*
    - Response: *“Alles klar, X23njv3ga. Hier ist dein Artikel.”*
  - Server speichert Daten in temp. Datei = “Session”
    - Server: `/tmp/sessions/X23njv3ga`, *“heute schon 2 Artikel”*
- Sitzungsverwaltung in PHP, u.a. über `$_SESSION`-Variable

Sessions in Symfony

- In einem Controller die Session benutzen:

```php
public function index(SessionInterface $session) {
    // Schreiben ....
    $session->set('foo', 'bar');
    // ... und Lesen
    $foobar = $session->get('foobar');
}
```

- Symfony kümmert sich darum, dass `$session` zum aktuellen Anfrager gehört
  - und unterstützt verschiedene Speicherorte (Datenbank, bestimmte Ordner)

Quelle: https://symfony.com/doc/current/controller.html#session-intro

Flash-Nachrichten

- Anwendungsfall:
  - Nutzer soll bei nächster Gelegenheit eine Nachricht sehen, egal, welche Seite er als nächstes sieht
    - z.B.: Formular zur Bearbeitung eines Blog-Artikels
    - zwei Optionen nach dem Speichern: *Nur speichern & weiter bearbeiten* sowie *Speichern & zurück zur Übersicht*
    - Erfolgsmeldung soll in jedem Fall angezeigt werden
- Lösung (Teil 1): Flash-Nachrichten; im Controller:
  -

```php
public function update(Request $request) {
    // ...
    if ($form->isSubmitted() && $form->isValid()) {
        // ...
        $this->addFlash('notice', 'Änderungen gespeichert.');
        return $this->redirectToRoute(...);
    }
}
```

Quelle: https://symfony.com/doc/current/controller.html#flash-messages

Flash-Nachrichten

- Lösung (Teil 2): Flash-Nachrichten; im Template

```
{% for message in app.flashes('notice') %}
  <div class="flash-notice">
    {{ message }}
  </div>
{% endfor %}
```

- Flash-Nachrichten werden in der Session gespeichert
  - und bleiben dort, bis sie abgerufen werden (siehe oben)
- Beispiel bei Supero:
  - Controller Admin::editHero()
  - Template admin/list_heros.html.twig

Quelle: https://symfony.com/doc/current/controller.html#flash-messages

Internationalisierung

Internationalisierung (oder: I18n)

- “internationalization” → “i” + 18 Zeichen + “n” → “i18n”
- je nach Website: verschiedene Sprach-Versionen nötig
- Symfony-Anwendungen unterstützen mehrere “Locales”
  - zu jedem Zeitpunkt gibt es genau ein aktives “Locale” (z.B. `en`)
    - Bestimmung z.B. durch URL-Teil, wie in
      - http://php.net/manual/de/language.basic-syntax.phptags.php, vs.
      - http://php.net/manual/en/language.basic-syntax.phptags.php
    - oder durch HTTP-Header `Accept-Language` (z.B. `de-DE`)
  - Locale kann im Session-Objekt gespeichert werden
    - damit ist es bei jedem Folge-Request schon bekannt
  - In Templates: Übersetzung-Funktionen aufrufen
    - anzuzeigende Texte werden aus Sprachdateien geladen (`translations/`)

Quelle: https://symfony.com/doc/current/translation.html

Sprach-Dateien

- Im Template:

```
{# Vorher #}
<h1>Hello, {{name}}</h1>
{# Jetzt #}
<h1>{% trans %}Hello, %name%{% endtrans %}</h1>
```

- Sprachdateien:

```
# translations/messages.de.yaml
'Hello, %name%': Hallo, %name%
Logout: Ausloggen
```

```
# translations/messages.fr.yaml
'Hello, %name%': Bonjour, %name%
Logout: Déconnecter
```

- Es gibt auch Unterstützung für komplexe Pluralformen

Quelle: https://symfony.com/doc/current/components/translation/usage.html

Zugriffskontrolle

Authentisierung/Autorisierung

- Was ist das?
  - Gibt es einen Unterschied?
- **Authentisierung** (wie “authentisch”)
  - Feststellen, *wer* jemand ist
  - z.B. durch Eingabe eines Passwortes
  - oder durch eine digitale Signatur (Public/Private-Key)
- **Autorisierung** (wie “Autorität”)
  - Feststellen, *was* dieser jemand tun darf
  - z.B. durch ein Regelwerk

Zugriffskontrolle in Symfony

- **Authentisierung**
  - per Konfiguration in `config/packages/security.yaml`
    - Referenz
- **Autorisierung**
  - teilweise in `config/packages/security.yaml`
  - flexibler: im Programmcode

Authentisierung in Symfony

- sehr flexibel, viele verschiedene Möglichkeiten, z.B.
  - HTTP Basic Authentication
    - (kennen wir schon)
  - Login-Formular
  - Authentisierung gegen LDAP-Server
  - Vorauthentisierte Nutzer (X.509, Kerberos)

Drei Fragen zur Authentisierung

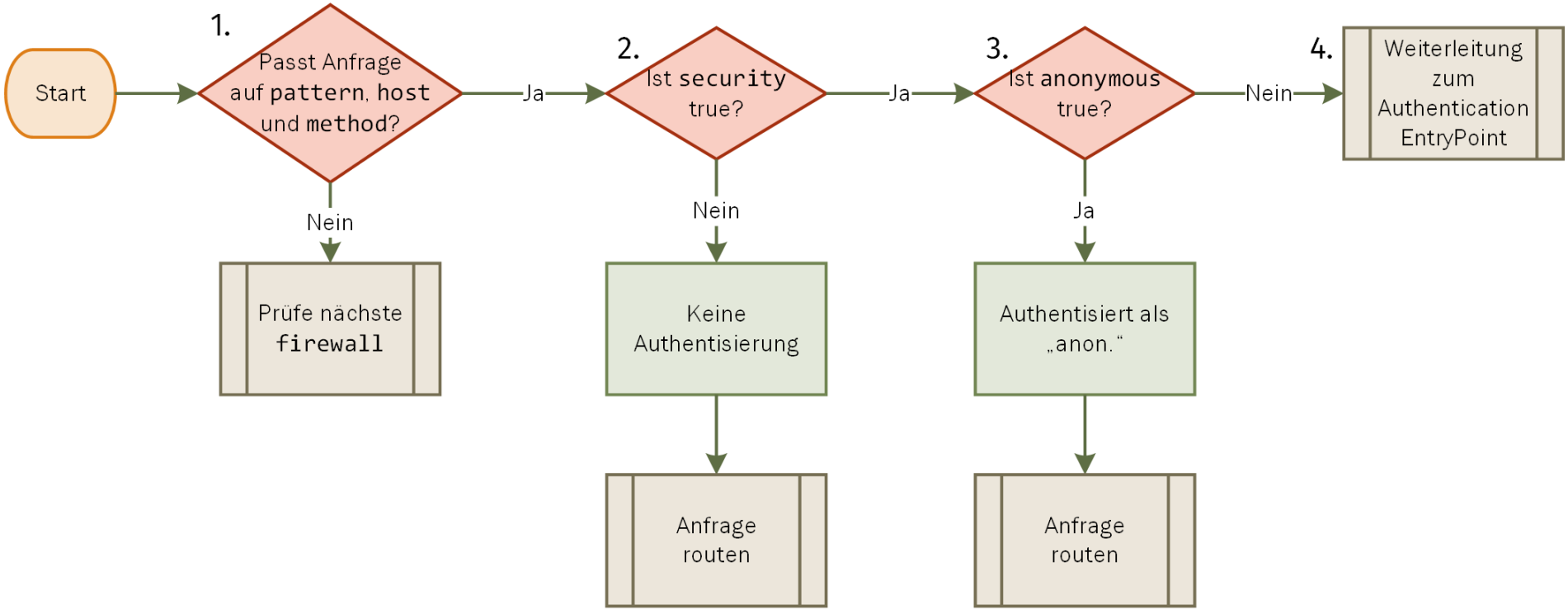
1. **Firewall**
   - Wie soll der Einlass geregelt sein?
     - HTTP Basic Auth vs. Login-Formular vs. vorauthentisiert vs. …
     - Kombinationen daraus
2. **Provider**
   - Wo kommen die Nutzer-Daten her?
     - Konfigurationsdatei vs. Datenbank vs. LDAP-Server vs. …
     - Kombinationen daraus
3. **Hashing** (in Symfony: “Encoding”)
   - Wie sind die Passwörter gespeichert?
     - Klartext vs. gehasht (z.B. BCrypt)

1. Firewall: Konfiguration

- Firewall in Symfony: Benannter Bereich (z.B. `main_area`)
  1. **Geltungsbereich**: Für welche Anfragen? (`pattern`, `host`, `method`)
     - Doku: How to Restrict Firewalls to a Specific Request
     - Default: jeweils `.*`, also aktiv für alle Anfragen
     - Erste zur Anfrage passende Firewall wird ausgewählt
  2. **Authentisierung**: an oder aus? (Flag: `security`)
     - Default: `security: true`
  3. **Anonymer Zugriff**: erlaubt oder nicht? (Flag: `anonymous`)
     - Default: `anonymous: false`
       - Achtung: anonyme Nutzer sind authentisiert – als `"anon."`
  4. Welche **Login-Methode**? (z.B. `http_basic`, `form_login`)
     - Default: *keine*
     - auch genannt: *AuthenticationEntryPoint*

1. Firewall: Konfiguration

- Der Weg durch eine Firewall



- *AuthenticationEntryPoint* wird aktiv:
  - wenn Firewall dies verlangt (siehe oben)
  - wenn bisherige Authentisierung nicht ausreicht (siehe später)

1. Firewall

- Zwei Login-Methoden:
  - HTTP Basic Authentication (Symfony-Doku)

```yaml
# config/packages/security.yaml
security:
  firewalls:            # Liste von Firewalls
    my_secure_area:     # Name der Firewall, selbst definiert
      http_basic: ~
```

  - Login-Formular (Symfony-Doku)

```yaml
security:
  firewalls:
    my_secure_area:     # selbst definiert
      form_login:
        login_path: my_login_route # Controller muss für diese
        check_path: my_login_route # Route(n) angegeben werden
```

2. Provider

- jeweils in `config/packages/security.yaml`
  - Nutzer aus der Konfig-Datei selbst (“Memory”)

```yaml
security:
  providers:                # Liste von Providern
    my_config_provider:     # Name eines selbst definierten Providers
      memory:
        users:              # Liste von Nutzern, hier nur einer
          my_username:      # Nutzer-Name
            password: ...       # (dazu gleich mehr)
            roles: 'ROLE_USER' # (dazu auch)
```

  - Nutzer aus Datenbank: braucht Entity-Klasse

```yaml
security:
  providers:
    my_db_provider:         # Name des Providers
      entity:
        class: App\Entity\MyUser # muss UserInterface implementieren
        property: username    # oder email, oder ...
```

3. Hashing (“Encoding”)

- jeweils in `config/packages/security.yaml`
  - Gar nicht (Klartext)

```yaml
security:
  encoders:             # Liste von Encodern
    My\User\Class:      # Pro verwendeter User-Klasse ein Encoder
      algorithm: plaintext
```

`passwort`

  - BCrypt:

```yaml
security:
  encoders:
    My\User\Class:
      algorithm: bcrypt
      cost: 12
```

`$2y$12$qO4jT2wcWYC1mjD6gh8f.OrGDGTqOetEW05.slZoxQCYsShKaros6`

Bsp: HTTP-Auth, Memory, Plaintext

```yaml
security:
  firewalls:
    my_secure_area:
      http_basic: ~                 # 1.
  providers:
    my_config_provider:
      memory:                       # 2.
        users:
          admin:                    # Nutzername: admin
            password: 'P4s5w0rT'    # Passwort:   P4sSw0rT
            roles: 'ROLE_USER'
  encoders:
    Symfony\Component\Security\Core\User\User: # Standard-Klasse
      algorithm: plaintext          # 3.
```

Bsp: Login, DB-Entity, Hashing

```
security:
  firewalls:
    my_secure_area:
      form_login:              # 1.
        login_path: login      # Route: bei fehlender Authentisierung
        check_path: login      # Route: Submit-Ziel für Formular
                     # Achtung: müssen selbst implementiert werden!
  providers:
    my_entity_provider:
      entity:                  # 2.
        class: App\Entity\User # muss UserInterface und Serializable
                               # implementieren
        property: username
  encoders:
    App\Entity\User:           # gleiche Klasse wie beim Provider
      algorithm: bcrypt        # 3.
      cost: 12
```

Authentisierung

- im Prinzip unabhäng: Firewall, Provider und Hashing
  - HTTP-Basic-Auth mit Abfrage einer Datenbank, in der Passwörter im Klartext liegen
  - Formular-Login gegen feste Nutzer-Daten, die gehasht in der `security.yaml` liegen
  - … oder was auch immer Sie sich ausdenken
- Wie und wann hashen?
  - Symfony: `if(hash(Nutzereingabe) == hinterlegter_Hash)`
    - d.h. nur bei Authentisierung
  - wie kommt aber der Hash in die Datenbank?
    - Programmatisch: Klartext mit `UserPasswordEncoderInterface` hashen
      - z.B. bei Nutzerregistrierung
    - Interaktiv: mit `php bin/console security:encode-password`
    - (beide verschlüsseln ensprechend der `security.yaml`)

Symfony-Doku zu Authentisierung

- Weil die Bestandteile unabhängig sind: Doku verstreut
  - Allgemeines
  - HTTP-Auth, Konfig-Provider, Hashing/kein Hashing
  - Login-Formular
    - Ausloggen
  - Entity-Provider (Nutzer aus DB)
  - Passwort-Hashing
- Eine Zusammenschau wie in den beiden Beispielen vorhin finden Sie dort leider nicht
  - Beispiel 1: HTTP Basic Auth, Memory-Provider, Klartext
  - Beispiel 2: Login-Formular, Entity-Provider, Hashing

Zugriffskontrolle in Symfony

- **Authentisierung** ✓
  - per Konfiguration in `config/packages/security.yaml`
    - Referenz
- **Autorisierung**
  - teilweise in `config/packages/security.yaml`
  - flexibler: im Programmcode

Autorisierung

- Verschiedene Modelle von Zugriffskontrolle, u.a.
  - **Rollenbasiert** (*Role-Based Access Control*)
    - Nutzer hat eine oder mehrere Rollen
      - z.B. *Nutzer*, *Moderator*, *Admin*
    - Rollen sind Rechte zugeordnet
      - z.B. *Blog-Artikel lesen* oder *Blog-Artikel löschen*
    - Anwender authentisiert sich als Nutzer, erhält dabei Rolle(n)
      - und damit die zugehörigen Rechte
      - z.B. *Peter* ist *Nutzer*, *Nutzer* darf *Blog-Artikel lesen*: *Peter* darf *Blog-Artikel lesen*
  - **Attributbasiert** (*Attribute-Based Access Control*)
    - feinkörniger als rollenbasierte Zugriffskontrolle
    - was ein Nutzer darf, ist je eine Einzelfall-Entscheidung
    - Grundlage: Attribute vom jeweiligen Geschäftsobjekt & Nutzer
      - z.B. *Peter darf nur seine eigenen Kommentare bearbeiten*

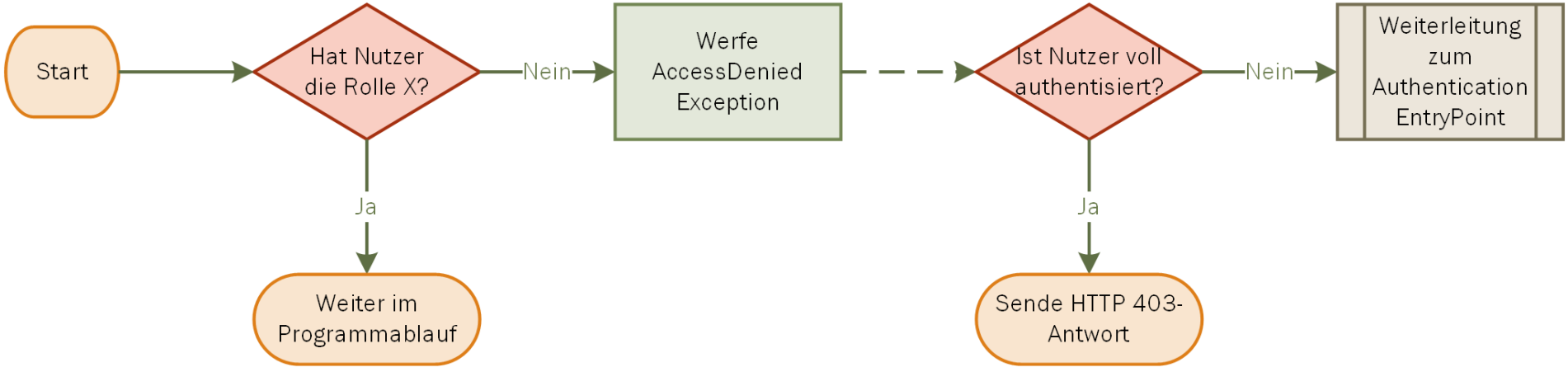
Rollenbasierte Zugriffskontrolle

- Woher kommen die Rollen?
  - Hängt vom Provider ab:
    - Memory-Provider: in Konfig-Datei, `users` > Name > `roles`
      - fest in der `security.yaml`-Datei
    - Entity-Provider: Implementierung von `UserInterface::getRoles()`
      - so flexibel Sie wollen
- Welche Rollen gibt es?
  - eine Rolle ist nur ein String; Sie definieren die selbst
  - Konvention: `ROLE_USER`, `ROLE_ADMIN`, `ROLE_SUPERSTAR`, …
- Wie erfolgt die Zugriffskontrolle?
  - URL-basiert, Controller-basiert, in Templates, …

Quelle: https://symfony.com/doc/current/security/entity_provider.html#what-s-this-userinterface

Rollenbasierte Zugriffskontrolle

- An verschiedenen Stellen: Rolle voraussetzen
  - URL-basiert vor Routing, vor Controller, im Controller
- Allgemeiner Ablauf in jedem Fall:



- *AuthenticationEntryPoint* wird aktiv:
  - wenn bisherige Authentisierung nicht ausreicht (siehe oben)
  - wenn Firewall dies verlangt (siehe vorhin)

Umsetzung: Rollenbasiert (1)

- URL-basiert (grob, per `access_control`):

```yaml
# config/packages/security.yaml
security:
    # ...
    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }
```

- Am Controller (feiner, per `IsGranted`-Annotation):

```php
// src/Controller/AdminController.php
/* ... */
public class AdminController {
  /** @IsGranted("ROLE_ADMIN") */
  public function listUsers() { /* ... */ }
}
```

Umsetzung: Rollenbasiert (2)

- Im Controller (beliebig fein im Kontrollfluss):

```php
// src/Controller/AdminController.php
/* ... */
public class AdminController extends Controller {
  public function listUsers() {
    /* ... */
    $this->denyAccessUnlessGranted('ROLE_ADMIN');
    /* ... */
  }
}
```

Quelle: https://symfony.com/doc/current/security.html#add-code-to-deny-access

Umsetzung: Rollenbasiert (3)

- Nur abfragen, ohne Auslöung von `AccessDeniedException`
  - Im Controller:

```php
// src/Controller/AdminController.php
/* ... */
public class AdminController extends Controller {
  public function listUsers() {
    if ($this->isGranted('ROLE_ADMIN')) { /* ... */ }
    /* ... */
  }
}
```

  - In Templates:

```twig
{% if is_granted('ROLE_ADMIN') %}
    <a href="...">Nutzer löschen</a>
{% endif %}
```

Quelle: https://symfony.com/doc/current/security.html#add-code-to-deny-access

Was wo kontrollieren?

- URL-basiert, per `security.yaml` > `access_control`:
  - sehr grob, dafür aber unabhängig von Routen
  - z.B. `path: ^/admin, roles: ROLE_ADMIN`
- Im Template:
  - Usability: keine falschen Möglichkeiten anbieten
    - fehlende Rechte lösen keine Authentisierung aus
- Im Controller:
  - Sicherheit: Prüfung *aller* Requests
    - fehlende Rechte können Authentisierung (bzw. HTTP 403) auslösen
  - Warum?
    - “Harmlos”: Nutzer könnte noch alte Seite im Browser haben
    - “Angriff”: HTTP-Request-Ziele (“Links”) kann man raten und Requests notfalls auch ohne Browser absenden
- Also: Prüfen Sie in den Templates UND im Controller

Autorisierung

- **Rollenbasiert** (*Role-Based Access Control*) ✓
  - Nutzer hat Rolle(n), Rollen haben Rechte, Nutzer erhalten Rechte über ihre Rolle(n)
- **Attributbasiert** (*Attribute-Based Access Control*)
  - feinkörniger als rollenbasierte Zugriffskontrolle
  - was ein Nutzer darf, ist je eine Einzelfall-Entscheidung
  - Grundlage: Attribute vom jeweiligen Geschäftsobjekt & Nutzer
    - z.B. *Peter darf nur seine eigenen Kommentare bearbeiten*

Umsetzung: Attributbasiert

- Symfony nutzt **Voter** für Zugriffs-Entscheidungen
  - bei jeder Kontrolle (im Controller oder Template, siehe oben): Voter stimmen ab über Zugriff
  - Symfony kennt Abstimmungsstrategien:
    - `affirmative`: Zugriff erlaubt, wenn einer der Voter zustimmt
    - `consensus`: Zugriff erlaubt, wenn die Mehrheit der Voter zustimmt
    - `unanimous`: Zugriff erlaubt, wenn alle Voter zustimmen
- attributbasierte Zugriffskontrolle: Voter implementieren

Quelle: https://symfony.com/doc/current/security/voters.html

Umsetzung: Voter implementieren

- Implementierung eines Voters
  - Klasse `Voter` erweitern
  - Methode `boolean supports($attr, $subject)` implementieren:
    - soll `true` liefern, wenn dieser Voter abstimmen möchte, wenn ein Zugriff der Art `$attr` auf `$subject` angefragt wird, z.B.

```
function supports($attr, $subject) {
  return ($subject instanceof BlogPostComment && $attr == 'edit');
}
```

  - Methode `boolean voteOnAttribute($attr, $subject, $token)`:
    - soll `true` leifern, wenn Zugriff durch `$token->getUser()` erlaubt ist, z.B.

```
function voteOnAttribute($attr, $subject, $token) {
  // $subject ist ein BlogPostComment
  // $attr ist 'edit'
  return ($subject->getOwner() === $token->getUser());
}
```

  - Das war's!

Quelle: https://symfony.com/doc/current/security/voters.html

Umsetzung: Voter befragen

- Im Template

```twig
{% if is_granted('edit', comment) %}
  {# Formular nur anzeigen, wenn Bearbeiten erlaubt ist #}
  {{ form(edit_comment_form) }}
{% endif %}
```

- Im Controller

```php
// src/Controller/BlogCommentController.php
/* ... */
public class BlogCommentController extends Controller {
  /** @Route("/blog/comment/{id}/edit") */
  public function editComment(BlogPostComment $comment) {
    $this->denyAccessUnlessGranted('edit', $comment);
    /* ... */
  }
}
```

Autorisierung

- **Rollenbasiert** (*Role-Based Access Control*) ✓
  - Nutzer hat Rolle(n), Rollen haben Rechte, Nutzer erhalten Rechte über ihre Rolle(n)
- **Attributbasiert** (*Attribute-Based Access Control*) ✓
  - feinkörniger als rollenbasierte Zugriffskontrolle
  - was ein Nutzer darf, ist je eine Einzelfall-Entscheidung
  - Grundlage: Attribute vom jeweiligen Geschäftsobjekt & Nutzer
    - z.B. *Peter darf nur seine eigenen Kommentare bearbeiten*

Hilfreich: `Security`

- Im Controller: `Security`-Objekt injizieren lassen
  - Authentisierung: `$security->getUser()` und `getToken()`
  - Autorisierung: `$security->isGranted()`
- Rückgabewerte bei verschiedenen Konfigurationen:

firewall in `security.yaml`	**`$security->get*()`**		**`$security->isGranted(…)`**
	User	**Token**	
`security: false`	`null`	`null`	*Exception*

firewall in `security.yaml` (`security: true`)	**`$security->get*()`**		**`$security ->isGranted('IS_AUTHENTICATED_*')`**	
	User	**Token**	**ANONYMOUSLY**	**FULLY**
`anonymous: true`	`null`	AnonymousToken	`true`	`false`
`http_basic: ~`	User	UsernamePasswordToken	`true`	`true`
`form_login:`	User	UsernamePasswordToken	`true`	`true`

Kommandozeile

Die Symfony-Konsole

- `php bin/console`
  - listet alle Befehle auf, die das aktuelle Symfony-Projekt unterstützt
- Meistens: aus den installierten Komponenten
- Einige kennen Sie schon, z.B.

```
php bin/console doctrine:database:create
php bin/console doctrine:schema:create
php bin/console doctrine:fixtures:load
```

- … andere noch nicht (es folgen Supero-Beispiele)

Routing-Konfiguration ansehen

```
$> php bin/console debug:router
 ------------------- -------- -------- ------ ----------------------
  Name                Method   Scheme   Host   Path
 ------------------- -------- -------- ------ ----------------------
  admin               ANY      ANY      ANY    /admin
  admin-list-heros    ANY      ANY      ANY    /admin/list-heros
  admin-add-hero      ANY      ANY      ANY    /admin/add-hero
  admin-edit-hero     ANY      ANY      ANY    /admin/edit-hero/{id}
  admin-delete-hero   ANY      ANY      ANY    /admin/delete-hero/{id
  buchen              ANY      ANY      ANY    /buchen/{name}
  start               ANY      ANY      ANY    /
  helden_liste        ANY      ANY      ANY    /helden
  helden_details      ANY      ANY      ANY    /helden/{name}
 ------------------- -------- -------- ------ ----------------------
```

Twig-Einstellungen ansehen

- Inklusive aller selbst definierten Funktionen und Filter

```
$> php bin/console debug:twig
...
Filters
-------
...
 * bootstrap
...
```

Syntax-Prüfung

- Twig-Templates (hier: alles im `templates`-Ordner)

```
$> php bin/console lint:twig templates

  [OK] All 12 Twig files contain valid syntax.
```

- YAML-Dateien (hier: alles im `config`-Ordner)

```
$> php bin/console lint:yaml config

  [OK] All 16 YAML files contain valid syntax.
```

Eigene Kommandos definieren

- Einiges kann/möchte man nicht über HTTP auslösen
  - z.B. das frisch Aufsetzen der Datenbank
- Symfony erlaubt eigene Kommandos zu implementieren
  - mit Hilfe-Seite, Parameter- und Options-Verarbeitung, …
- Beispiel in Supero:
  - `php bin/console supero:reset-database`
  - Implementierung: `src/Command/ResetDatabaseCommand.php`
- Kommandos haben auf praktisch alles Bekannte Zugriff
  - (außer z.B. das Request-Objekt und die Authentisierung)
  - auf Doctrine, das Dateisystem, die Konfigurationsdateien, …

Quelle: https://symfony.com/doc/current/console.html

Debugging

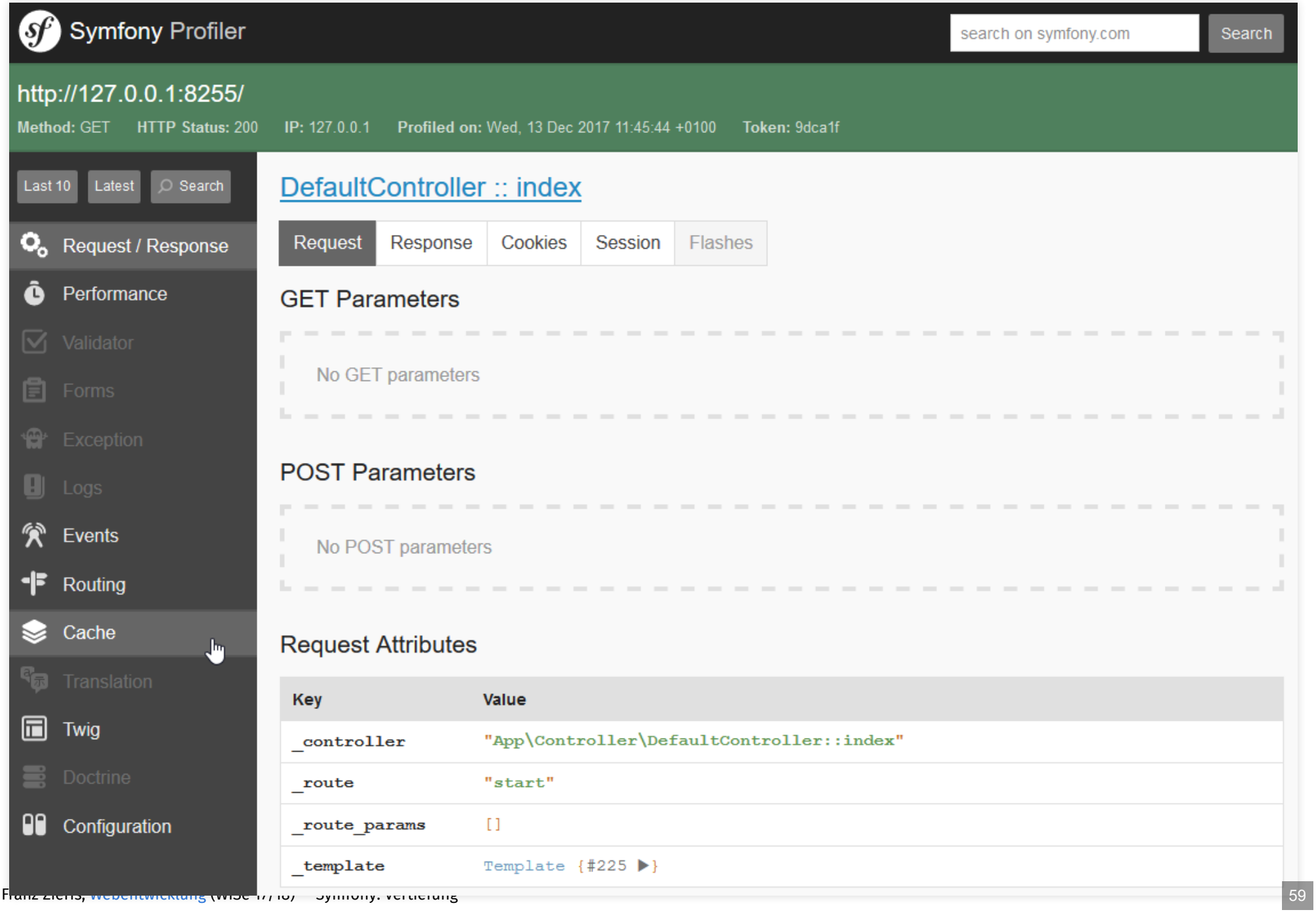
Debugging

- *Profiler*, das PHP-Gegenstück zu `F12`

```
$> composer require --dev profiler
```

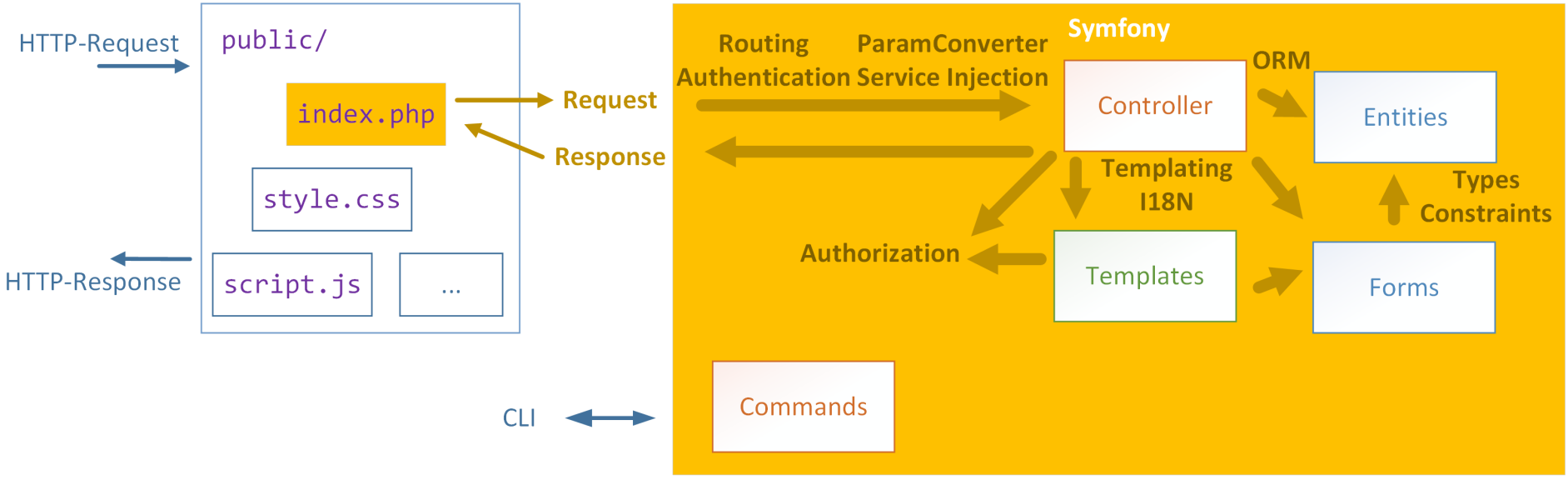
- Beim Aufruf einer Seite:

Quelle: https://symfony.com/doc/...#the-web-debug-toolbar-debugging-dream


Symfony Profiler
search on symfony.com
Search
http://127.0.0.1:8255/
Method: GET HTTP Status: 200 IP: 127.0.0.1 Profiled on: Wed, 13 Dec 2017 11:45:44 +0100 Token: 9dca1f
Last 10
Latest
Search
Request / Response
Performance
Validator
Forms
Exception
Logs
Events
Routing
Cache
Translation
Twig
Doctrine
Configuration
DefaultController :: index
Request
Response
Cookies
Session
Flashes
GET Parameters
No GET parameters
POST Parameters
No POST parameters
Request Attributes
Key
Value
_controller
"App\Controller\DefaultController::index"
_route
"start"
_route_params
[]
_template
Template {#225 ▶}

Zusammenfassung: Symfony

- Symfony-Architektur:



- Außerdem:
  - Funktionsweise der Firewall und Auslösen der Authentisierung
  - Rollenbasierte und attributbasierte Autorisierung

Danke!