Webentwicklung

Backend: Symfony

Inhalt dieser Einheit

1. Das Symfony-Framework
2. Routing
3. View
4. Model
5. Formulare

Letzte Einheit

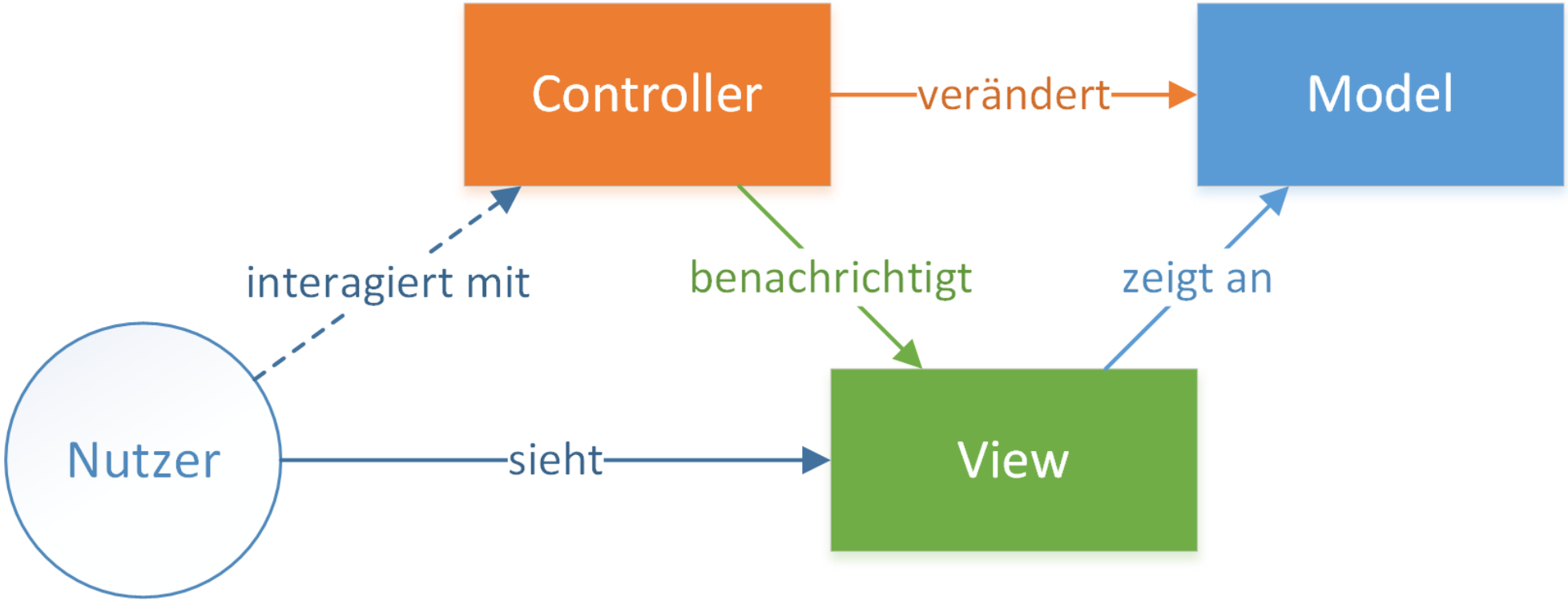
- Wiederverwendung von Komponenten für …
  - Trennung von Geschäftslogik und Darstellung:
    - Templates mit Twig
  - Trennung von Geschäftslogik und Datenhaltung:
    - Objekt-Relationaler Mapper (ORM): Doctrine

Das Symfony-Framework

MVC: Model View Controller

- klassisches Entwurfsmuster
  - Ergänzung des Observer-Patterns
- Idee:
  - Daten auf verschiedene Weisen betrachten/bearbeiten können
  - automatische Aktualisierung aller Anzeigen
- Teile:
  - **Model**: Sorgt für die Datenhaltung
  - **View**: Zeigt den aktuellen Stand des Models an
  - **Controller**: Koordiniert Änderungen am Model und benachrichtigt alle Views (Observer-Pattern)
- zentral in der Entwicklung von grafischen Desktop-Anwendungen

MVC: Schematisch

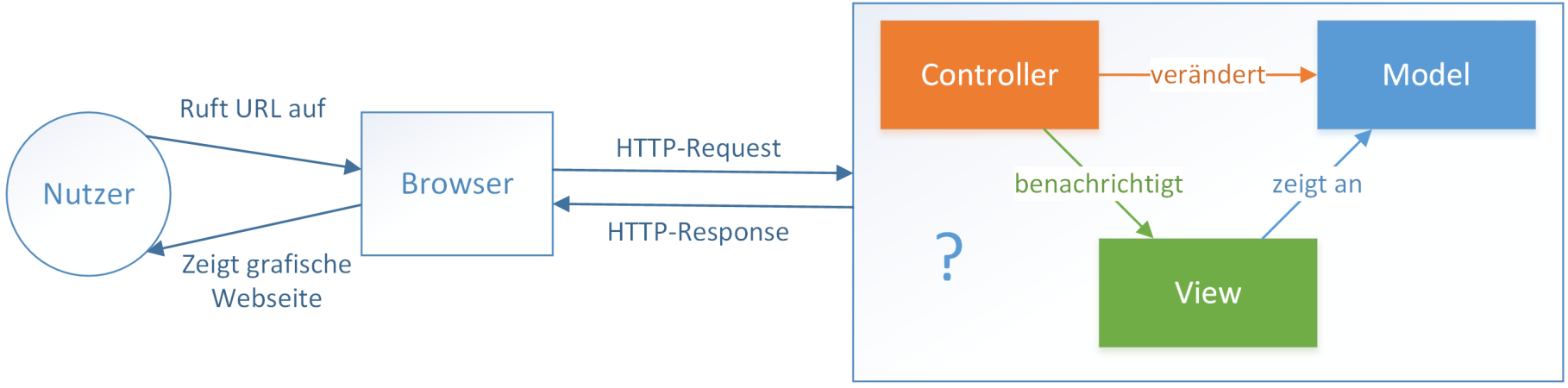


MVC im Web

- bekannter früher Vertreter: Ruby on Rails
  - viele Ideen in PHP übernommen: Symfony 1
- **Framework** (= *Rahmenwerk*)
  - stellt *Struktur* und Hilfsfunktionen bereit
  - eigentliche Funktion muss noch implementiert werden
  - im Ggs. zu Bibliothek/Komponenten: in sich abgeschlossen

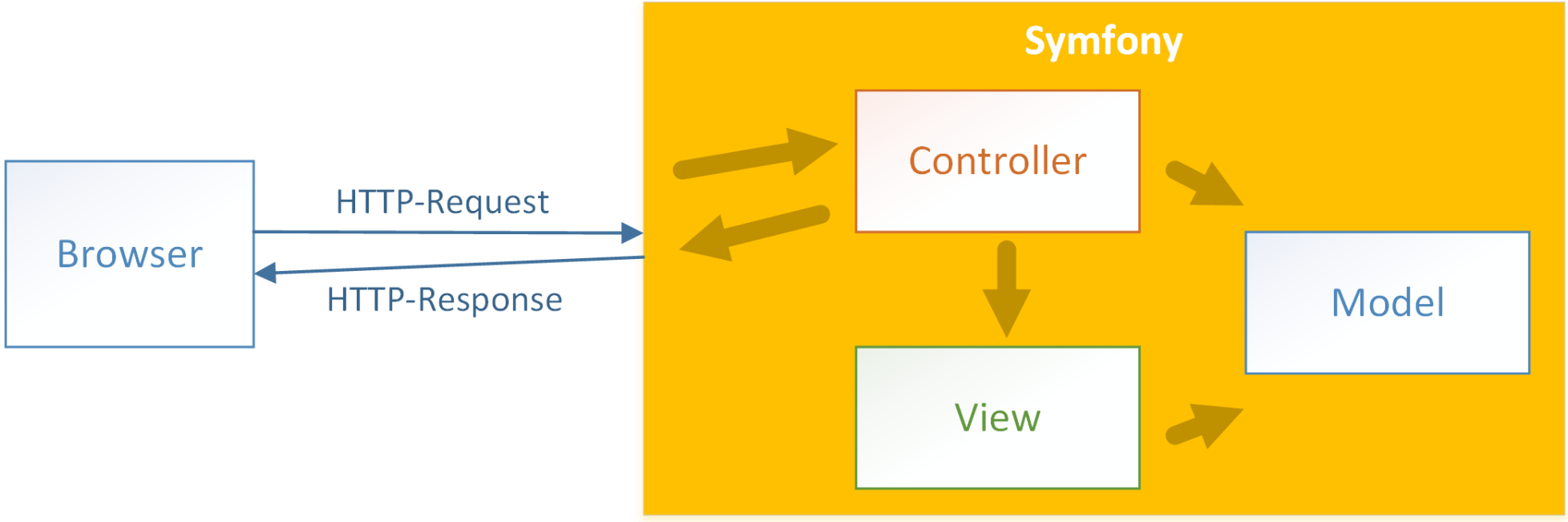
MVC im Web: Schematisch

- Aber: Web != Desktop
  - Wie kommt der Request zum Controller?
  - Wenn die View passiv ist, wer erzeugt die Response?



Symfony-Architektur

- “Request/Response Framework”
  - zentral: Controller erhalten Request und erzeugen Response
  - Model und View können dabei helfen



Erster Controller

- Ausgangspunkt: Symfony frisch installiert
- Controller: `src/Controller/HelloController.php`

```php
namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class HelloController {
  /**
   * @Route("/hallo")
   */
  public function greet() {
    return new Response("Hallo Welt");
  }
}
```

- Symfony routet Anfragen an `/hello` an `greet`-Methode
  - Antwort auf Anfrage `GET /hello`: `Hallo Welt`

Steuerung der Server-Antwort

```php
/* src/Controller/HelloController.php */
class HelloController {
  /**
   * @Route("/hallo")
   */
  public function greet() {
    return new Response("'Welt' nicht gefunden", 404);
  }
}
```

- Symfony erzeugt HTTP-Antwort mit Status-Code `404`

Controller

- In Symfony: *Controller* …
  - ist Ziel einer **Route**,
  - nimmt **Request** entgegen, und
  - erzeugt eine **Response**.
- Technisch: eine PHP-Funktion
  - normalerweise: Methode einer `*Controller`-Klasse
    - (Symfony ist da aber flexibel)
- Typisch: Erweitern von Symfonys `Controller`-Klasse
  - viele nützliche Hilfsfunktionen

Symfony-Controller-Klasse

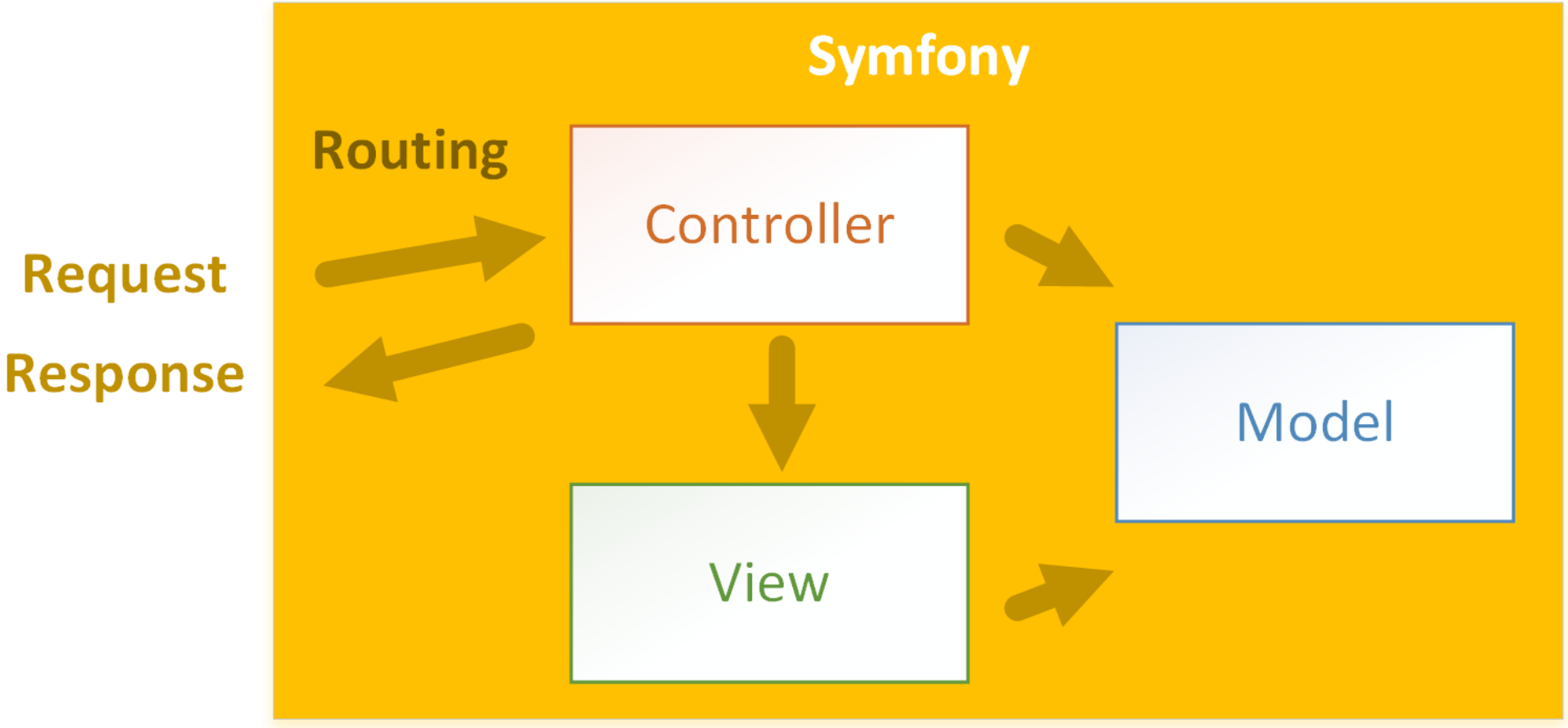
```php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class SomeController extends Controller {
    /**
     * @Route("/htw")
     */
    public function htw() {
        // Sendet HTTP 302 an Browser
        return $this->redirect("http://www.htw-berlin.de");
    }

    /**
     * @Route("/super-secret")
     */
    public function forbidden() {
        // Sendet HTTP 403 an Browser
        throw $this->createAccessDeniedException();
    }
}
```

Routing

Routing

- Routing: Abbildung eines URL-Path auf einen Controller
  - Lesbare URLs:
    - Nicht: `mein-blog.de/index.php?artikel_id=157`
    - Besser: `mein-blog.de/erste-schritte-mit-symfony`
  - Stabile URLs: (“Cool URIs don’t change”)
    - aktuell: `index.php` und `artikel_id`
    - in 6 Monaten: `article.php` und `id`
  - Leicht änderbare URLs:
    - wenn dann *doch* mal eine Änderung nötig ist:
      Will man wirklich manuell alle `<a href="...">` durchgehen?
- In Symfony: verschiedene Möglichkeiten
  - Annotationen am Controller
  - Separate Datei: `routes.yaml`, `routes.xml` oder `routes.php`

Routing in der Architektur



Route mit Parametern

```php
/* src/Controller/HelloController.php */
class HelloController {
  /**
   * @Route("/hallo/{name}")
   */
  public function greet($name) {
    return new Response("Hallo $name");
  }
}
```

- Symfony liest HTTP-Anfrage und bestimmt Parameter für Controller-Aufruf

Pattern-Matching in URLs

```php
/* src/Controller/HelloController.php */
class HelloController {
  /**
   * @Route("/hallo/{name}", requirements={"name": "[^\d]+"})
   */
  public function greet($name) {
    return new Response("Hallo $name");
  }

  /**
   * @Route("/hallo/{anzahl}", requirements={"anzahl": "\d+"})
   */
  public function repeatHello($anzahl) {
    return new Response(str_repeat("Hallo Welt\n", $anzahl));
  }
}
```

- `\d` steht im regulären Ausdruck für “Digit”, also `0-9`

Benannte Routen

```php
/* src/Controller/HelloController.php */
class HelloController extends Controller {
  /** @Route("/tschuess", name="abschied") */
  public function sayGoodbye() { /* ... */ }

  /** @Route("/ciao) */
  public function sayCiao() {
    // HTTP 302 an '/tschuess'
    return $this->redirectToRoute('abschied');
  }

  /** @Route("/hallo/{name}", name="gruesse") */
  public function greet($name) { /* ... */ }

  /** @Route("/klaus") */
  public function halloKlaus() {
    // HTTP 302 an '/hallo/Klaus'
    return $this->redirectToRoute('gruesse', ['name' => 'Klaus']);
  }
}
```

View (mit Templates)

Auftritt eines alten Bekannten: Twig

Einbindung von Templates

- Twig ist nach der Installation bereits konfiguriert

```php
/* src/Controller/HelloController.php */
class HelloController extends Controller {
  /**
   * @Route("hallo/{name}", name="gruesse")
   */
  public function greet($name) {
    return $this->render("hello/greet.html.twig", [
      'name' => $name
    ]);
  }
}
```

```twig
{# templates/hello/greet.html.twig #}
Hallo {{ name }}!
```

- `$this->render()` erzeugt ein `Response`-Objekt

Haupt- und Nebensachen trennen

- Annotation `@Template` spart den `render()`-Aufruf

```php
/* src/Controller/HelloController.php */
class HelloController extends Controller {
  /**
   * @Route("hallo/{name}", name="gruesse")
   * @Template("App:Hello:greet.html.twig")
   */
  public function greet($name) {
    return ['name' => $name];
  }
}
```

```twig
{# templates/hello/greet.html.twig #}
Hallo {{ name }}!
```

- Rückgabe: Keine `Response`, sondern Daten-Array fürs Template
  - Symfony erkennt das, und erstellt selbst die `Response`

Konvention über Konfiguration

- Wenn Template- und Controllername passen:

```php
/* src/Controller/HelloController.php */
class HelloController extends Controller {
  /**
   * @Route("hallo/{name}", name="gruesse")
   * @Template
   */
  public function greet($name) {
    return ['name' => $name];
  }
}
```

```twig
{# templates/hello/greet.html.twig #}
Hallo {{ name }}!
```

- Ist kein Templatename angegeben, rät Symfony
  - Namenskonvention einhalten: keine Konfiguration nötig

Konvention über Konfiguration (2)

- Es geht noch kürzer:

```php
/* src/Controller/HelloController.php */
class HelloController extends Controller {
  /**
   * @Route("hallo/{name}", name="gruesse")
   * @Template
   */
  public function greet($name) {
    // LEER. Hier gibt es nichts zu tun!
  }
}
```

```twig
{# templates/hello/greet.html.twig #}
Hallo {{ name }}!
```

- Controller ohne Rückgabewert:
  - Symfony reicht Parameter aus Signatur weiter
    - (an das Template mit dem passenden Namen)

Template-Helfer

- Symfony erweitert die Twig-Funktionalität
  - z.B. für benannte Routen:

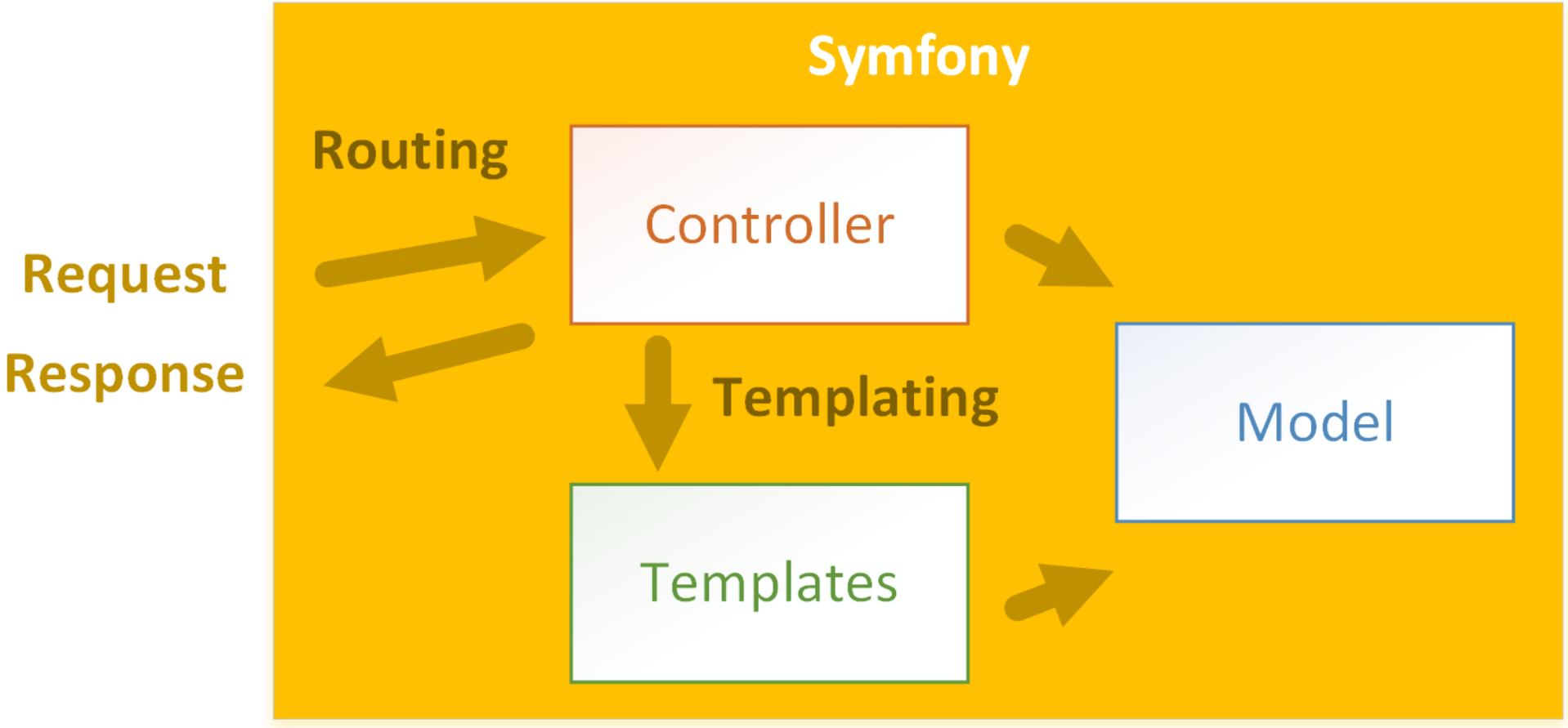
```
<!-- Ziel-Adresse selbst hinschreiben -->
<a href="/hallo/Klaus">Klaus grüßen</a>

<!-- oder aus Routennamen und Parameter generieren lassen -->
<a href="{{ path('gruesse', {'name': 'Klaus'}) }}">nochmal</a>
```

  - oder für einzubindende Ressourcen (“Assets”)

```
<link href="{{ asset('css/main.css') }}" rel="stylesheet" />
```

    - das erlaubt z.B. alle statischen Ressourcen auf anderen Servern zu speichern, ohne dass die Templates davon wissen müssen
  - (weitere Beispiele folgen auf späteren Folien)

Templating in der Architektur



Model (mit Entities)

Noch ein alter Bekannter: Doctrine

Einbindung des ORM

```php
/* src/Controller/HelloController.php */
class HeroController extends Controller {
  /**
   * @Route("/helden/{name}")
   * @Template
   */
  public function steckbrief($name) {
    $hero = $this->getDoctrine()
                 ->getRepository(Hero::class)
                 ->findOneBy(['name' => $name]);
    return ['hero' => $hero];
  }
}
```

- Abrufen eines Helden anhand des Namens
  - Doctrine und den EntityManager kennen wir schon
- Übergabe des `Hero`-Objekts an Template
  - (Template-Name per Konvention: `hero/steckbrief.html.twig`)

Haupt- und Nebensachen trennen

```php
/* src/Controller/HelloController.php */
class HeroController extends Controller {
  /**
   * @Route("/helden/{name}")
   * @Template
   * @ParamConverter("hero", class="App:Hero")
   */
  public function steckbrief(Hero $hero) {
    return ['hero' => $hero];
  }
}
```

- `@ParamConverter`:
  - nutzt Platzhalter aus Route (hier `name`)
  - um ein eindeutiges `Hero`-Objekt abzufragen
  - und der Controller-Methode zu übergeben
  - (erzeugt HTTP Status 404 falls keiner gefunden wurde)

Konvention über Konfiguration

```php
/* src/Controller/HelloController.php */
class HeroController extends Controller {
  /**
   * @Route("/helden/{name}")
   * @Template
   */
  public function steckbrief(Hero $hero) {
    return ['hero' => $hero];
  }
}
```

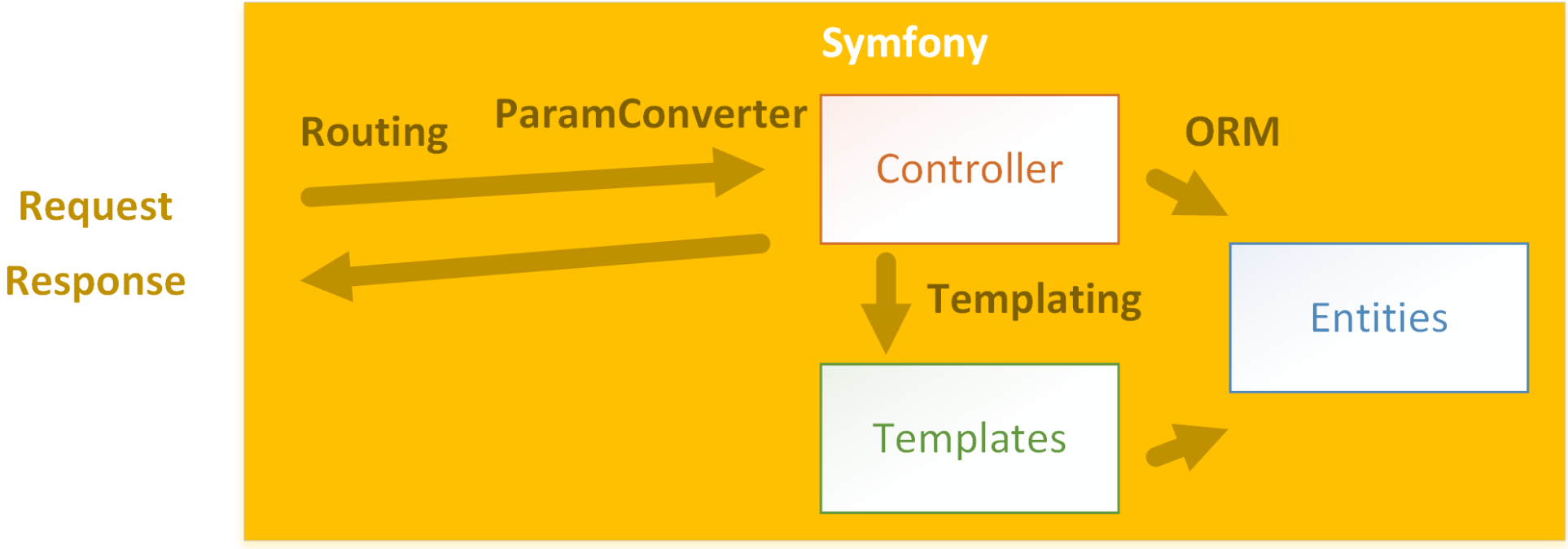
- Signatur verlangt nach einem `Hero`-Objekt
  - mehr muss der `ParamConverter` nicht wissen
  - hier ist nicht mal eine Annotation nötig!
    - (`ParamConverter` ist in Request-Vorverarbeitung eingebunden)

Konvention über Konfiguration (2)

```php
/* src/Controller/HelloController.php */
class HeroController extends Controller {
  /**
   * @Route("/helden/{name}")
   * @Template
   */
  public function steckbrief(Hero $hero) {
    // LEER. Hier gibt es nichts zu tun!
  }
}
```

- Noch einfacher: alle Template-Parameter in Signatur
  - `@Template` übergibt Parameter direkt an Template
    - (den Trick kannten wir schon)

ORM in der Architektur



Controller-Signatur

- Die Signatur des Controllers ist mächtig
  - Man sagt Symfony was man braucht, Symfony kümmert sich
    - (“Dependency Injection”, Abhängigkeiten werden injiziert)
- Beispiele:
  - Das Anfrage-Objekt:

```
public function foo(Request $request) { ... }
```

  - Die Nutzer-Session:

```
public function foo(SessionInterface $session) { ... }
```

  - Doctrines EntityManager:

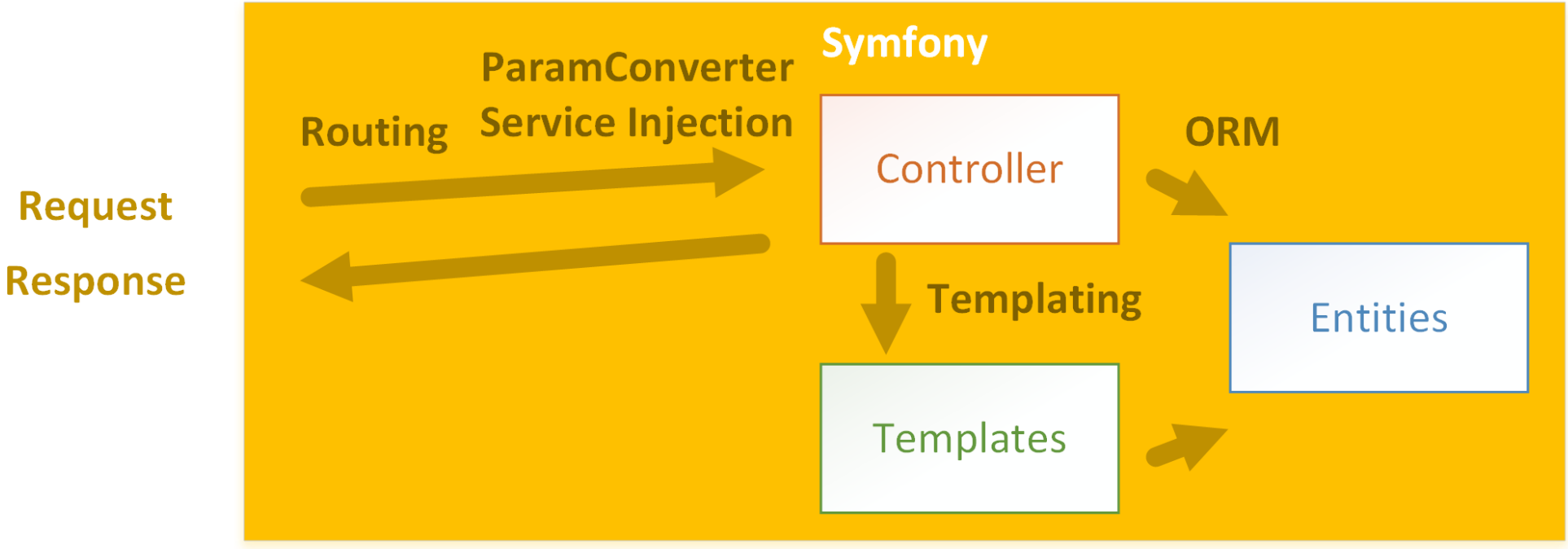
```
public function foo(EntityManagerInterface $em) { ... }
```

  - Logger (vgl. Log4J)

```
public function foo(LoggerInterface $logger) { ... }
```

- und viele mehr (in beliebigen Kombinationen)

Services in der Architektur



Formulare

Interaktion für Webanwendungen

Formulare händisch bauen

- Formulare kann man zwar händisch bauen …
  - Im Template:

```
<form method="post" action="{{ path('kontakt') }}">
  <input type="text" name="email">
  <!-- ... -->
</form>
```

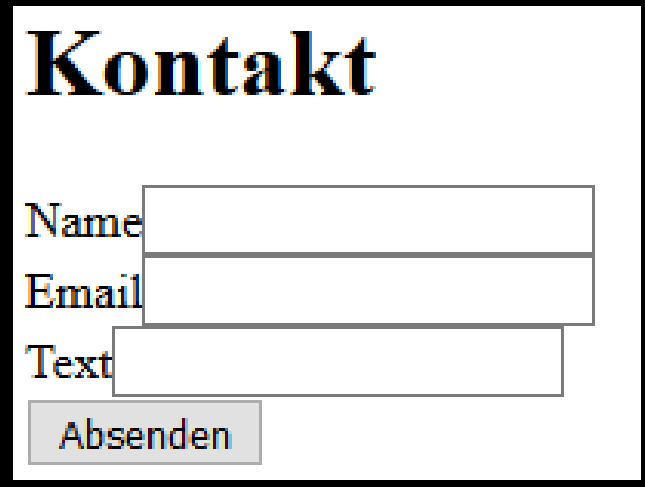
  - Im Controller:

```
$email = $request->get('email');
if (!(preg_match('/\w+@\w+\.\w+/', $email))) {
  // ungültige E-Mail-Adresse
}
// ...
```

- … aber:
  - Formular ist so nicht leicht wiederverwendbar (`action=?`)
  - Werte-Validierung für jedes Formular schreiben/einbinden
  - Änderung von Datentypen: Controller und Template anpassen

Formulare programmatisch bauen

```php
/* src/Controller/ContactController.php */
/**
 * @Route("/kontakt")
 * @Template
 */
public function contact() {
  $form = $this->createFormBuilder()
               ->add('name')
               ->add('email')
               ->add('text')
               ->add('absenden', SubmitType::class)
               ->getForm();
  return ['kontakt' => $form->createView() ];
}
```




```twig
<h1>Kontakt</h1>
{{ form(kontakt) }}
```

- `form`: Twig-Hilfsfunktion, erzeugt HTML-Formular
  - mit aufrufendem Controller als Ziel (`action="..."`)

Formular-Daten behandeln

```php
public function contact(Request $request) {
  $form = $this->createFormBuilder()/* ... */->getForm();

  // Formular-Informationen aus Request extrahieren
  $form->handleRequest($request);
  // wenn Formular abgesendet wurde (= 2. Controller-Aufruf)
  if ($form->isSubmitted()) {
    $data = $form->getData(); // assoz. Array
    return Response("Formular wurde abgesendet");
  }

  // sonst (= 1. Controller-Aufruf)
  return ['kontakt' => $form->createView() ];
}
```

- entscheidend: `handleRequest()` und `isSubmitted()`
- (Echte Formularbehandlung: E-Mail Versenden, Datenbankeintrag machen, …)

Verschiedene Feldtypen

- *Viele* eingebaute Typen:
  - `Text` (Default), `Textarea`, `Email`, `Integer`, `Money`, `Number`, `Password`, `Percent`, `Search`, `Url`, `Range`, `Tel`, `Color`
  - `Choice`, `Entity`, `Country`, `Language`, `Locale`, `Timezone`, `Currency`
  - `DateType`, `DateInterval`, `DateTime`, `Time`, `Birthday`,
  - `Checkbox`, `File`, `Radio`, `Collection`, `RepeatedType`, `HiddenType`
  - `Button`, `Reset`, `Submit`
    - natürlich durch eigene Typen erweiterbar
- je mit diversen Optionen für flexiblen HTML-Ouput
  - z.B. `Choice`: `select`-Tag, Radio-Buttons, Checkboxes

Validierung

- *Viele* eingebaute Constraints, z.B.:
  - `NotBlank`, `Blank`, `NotNull`, `IsNull`, `IsTrue`, `IsFalse`, `Type`
  - `Email`, `Length`, `Url`, `Regex`, `Ip`, `Uuid`
  - `Range`, `EqualTo`, `NotEqualTo`, `IdenticalTo`, `NotIdenticalTo`, `LessThan`, `LessThanOrEqual`, `GreaterThan`, `GreaterThanOrEqual`
  - `Date`, `DateTime`, `Time`, `Choice`, `Collection`, `Count`, `UniqueEntity`, `Language`, `Locale`, `Country`
  - `File`, `Image`, `Bic`, `CardScheme`, `Currency`, `Luhn`, `Iban`, `Isbn`, `Issn`
    - natürlich durch eigene Constraints erweiterbar
- Unterschied zu Feldtypen:
  - Feldtypen sind für die Darstellung im Browser
    - evtl. mit clientseitiger Validierung, `<input type='email'>`
  - Constraints sind für die Verarbeitung im Controller
    - serverseitige Validierung, `new Constraint\Email()`

Validierung: Beispiel (1/2)

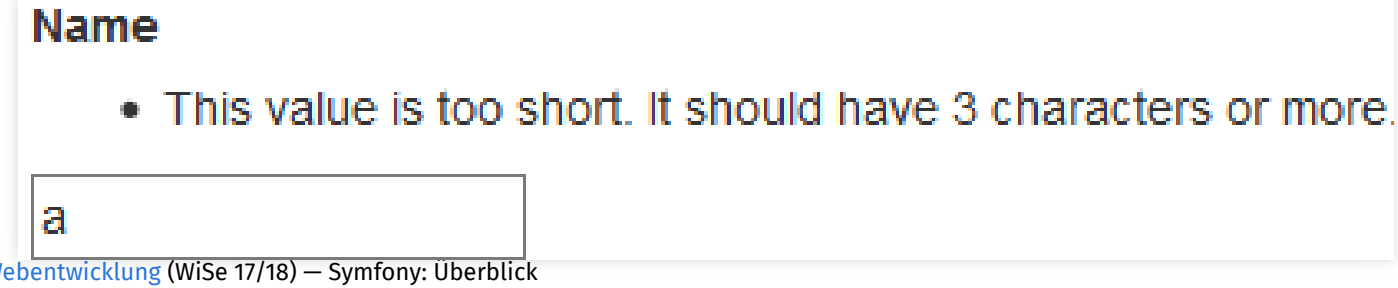
```
$form = $this->createFormBuilder()
  ->add('name', TextType::class, [
    'constraints' => [new Length(['min' => 3])]
  ])
  ->add('email', EmailType::class, [
    'constraints' => [new Email()]
  ])
  ->add('text', TextAreaType::class, [
    'constraints' => [new Length(['max' => 500])]
  ])
  ->add('absenden', SubmitType::class)
  ->getForm();
```

- optional drei Parameter von `add()`:
  1. Feld-Name (String)
  2. Feld-Typ (Klasse)
  3. Feld-Optionen (Assoziatives Array)
     - Key `constraints`: Liste (Array) von Constraints

Validierung: Beispiel (2/2)

```php
$form = $this->createFormBuilder()
        /* Definition mit Constraints ... */
        ->getForm();

$form->handleRequest($request);
if ($form->isSubmitted() && $form->isValid()) { // neu: isValid()
  $data = $form->getData();
  return Response("Formular wurde abgesendet");
}

// sonst: Formular wieder anzeigen (inkl. Fehlermeldungen)
return ['kontakt' => $form->createView() ];
```

- entscheidend: `isValid()`
  - Template: Anzeige aller Fehler (evtl. mehrere pro Feld)



Formulare für Model-Klassen

Formular für Model-Klassen

```php
/** @Route("edit-hero/{id}") */
public function editHero(Request $request, Hero $hero) {
  // hier: $hero automatisch durch {id} aus URL ermittelt
  $form = $this->createFormBuilder($hero) // <-- neu
    ->add('name')->add('price')
    /* ... weitere Felder ... */
    ->add('save', SubmitType::class)
    ->getForm();

  $form->handleRequest($request);
  // hier: $hero mit Formularwerten aktualisiert

  if ($form->isSubmitted() && $form->isValid()) {
    // Alles ok, $hero-Objekt kann man speichern, z.B. mit:
    //   $doctrine->persist($hero);
    //   $doctrine->flush();
    return $this->redirect(...);
  }
  // Formular (nochmal) anzeigen
  return $this->render(...);
}
```

Modellierung in Model-Klassen

- `FormBuilder` bekommt nun “Start-Werte” durch `$hero`

```
$form = $this->createFormBuilder($hero)->...
```

- dadurch: kennt Model-Klasse und kann sinnvoll raten
  - Feldtypen anhand der Model-Column-Typen, z.B.:
    - `@Column(type="text")` ⟶ `TextareaType::class`
    - `@Column(type="boolean")` ⟶ `CheckboxType::class`
  - Feld-Constraints anhand von Model-Constraints, z.B.:
    - `@Assert\Email` ⟶ `'constraints' => [new Email()]`
- Idee: *Model*-Klassen zur *Modellierung* benutzen
  - d.h. viel über Datentypen und Constraints ausdrücken
    - statt über explizite (und verstreute) Prüf-Logik

Best Practice: Formular-Klassen

```php
class HeroType extends AbstractType {
  public function buildForm(FormBuilderInterface $builder /* */) {
    $builder->add('name')/* ... weitere Felder ... */;
  }
  public function configureOptions(OptionsResolver $resolver) {
    $resolver->setDefaults(['data_class' => Hero::class]);
  }
}
```

```php
// Formular-Klasse im Controller benutzen
public function editHero(Request $request, Hero $hero) {
  $form = $this->createForm(HeroType::class, $hero);
  // ...
}
public function addHero(Request $request) {
  $hero = new Hero();
  $form = $this->createForm(HeroType::class, $hero);
  // ...
}
```

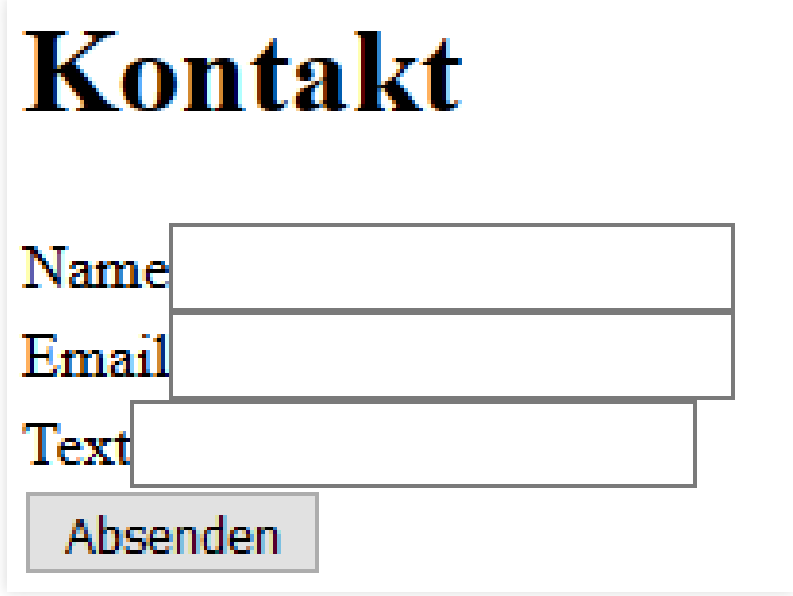
Quelle: https://symfony.com/doc/current/best_practices/forms.html

Optische Gestaltung der Formulare

- Schön geht anders …

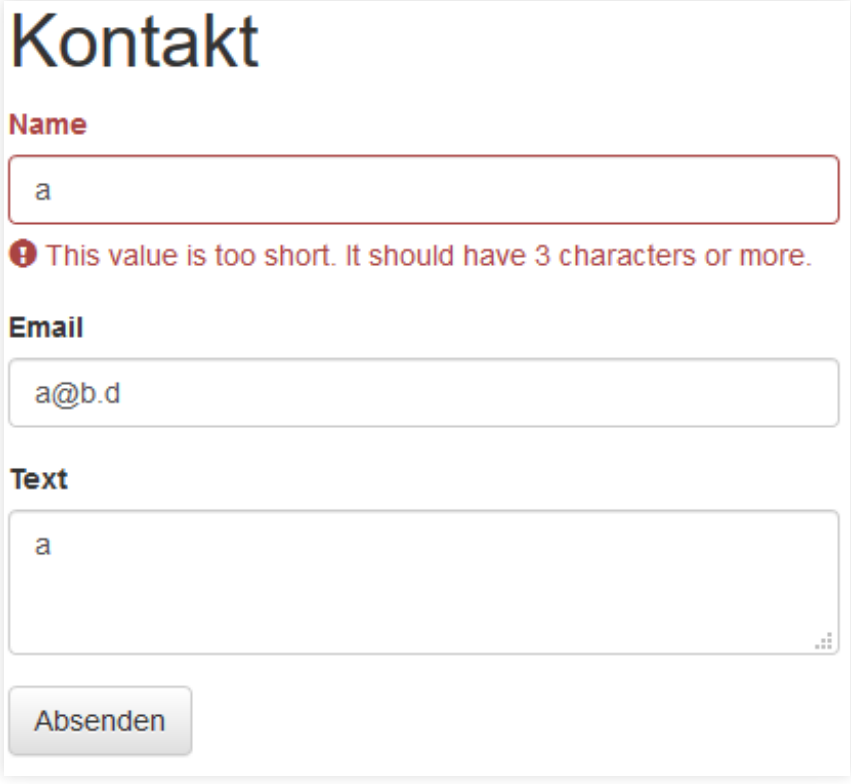
```
<h1>Kontakt</h1>
{{ form(kontakt) }}
```

- Twig-Helper `form()`:
  - kürzeste Schreibweise
  - benutzt Standard-Theme
- beides lässt sich ändern



Einfach: Anderes Theme benutzen

```
{% form_theme kontakt 'bootstrap_3_layout.html.twig' %}
<h1>Kontakt</h1>
{{ form(kontakt) }}
```



- Standard-Theme für alle Formulare: per Konfiguration

Mehr Kontrolle über Darstellung

- `form(kontakt)` ruft intern auf:
  - `form_start(kontakt)`: passender `<form>`-Tag
  - `form_widget(kontakt)`: alle definierten Felder
  - `form_end(kontakt)`: (alle nicht gerenderten Felder und) `</form>`
- `form_widget(kontakt)` geht über alle Felder und ruft auf:
  - `form_label(feld)`: Name des Feldes
  - `form_errors(feld)`: Liste evtl. Fehler
  - `form_widget(feld)`: das eigentliche Feld
- Funktionen im Template direkt aufrufbar
  - z.B. in einer HTML-Tabelle →

Beispiel: Volle Kontrolle

```
{{ form_start(kontakt) }}
<table>
  <tr>
    <td>{{ form_label(kontakt.name) }}</td>
    <td>{{ form_widget(kontakt.name) }}<br>
        {{ form_errors(kontakt.name) }}</td>
  </tr>
  <!-- ... -->
</table>
{{ form_end(kontakt) }}
```

- Wenn man sowas häufiger macht: eigenes Theme

Zusammenfassung: Formulare

- sehr umfangreicher und mächtiger Bereich in Symfony
- Merke:
  - Niemals Formulare selbst im Template zusammenbauen
    - immer einen `FormBuilder` nutzen
    - Best Practice: Formular als eigene Klasse
  - Wenn es eine Model-Klasse (“Entity”) gibt: benutzen
    - Dann kann der `FormBuilder` die Feld-Typen und Validierer “raten”
  - Validierung:
    - Feld-Typen für **client-seitige** Validierung (`input type=email`)
    - Constraints für **server-seitige** Validierung (`new Constraints\Email()`)
  - Optik:
    - Twig-Helfer benutzen, um Formular “aufzubrechen” (`form_*()`)
    - Themes benutzen für konsistente Formular-Optik

Symfony kann noch viel mehr …

- Nächste Einheit:
  - Sitzungsverwaltung
  - Sicherheit und Zugriffsschutz
- Später:
  - Responses jenseits von HTML
  - Interaktion mit JavaScript (AJAX)
  - Caching, Performance

Informationen zu Symfony

- Webseite: https://symfony.com
  - *vieeel* Dokumentation
- Symfony 4 ist kein “Monolith”, sondern “Micro”:
  - starte minimal, ergänze Komponenten bei Bedarf

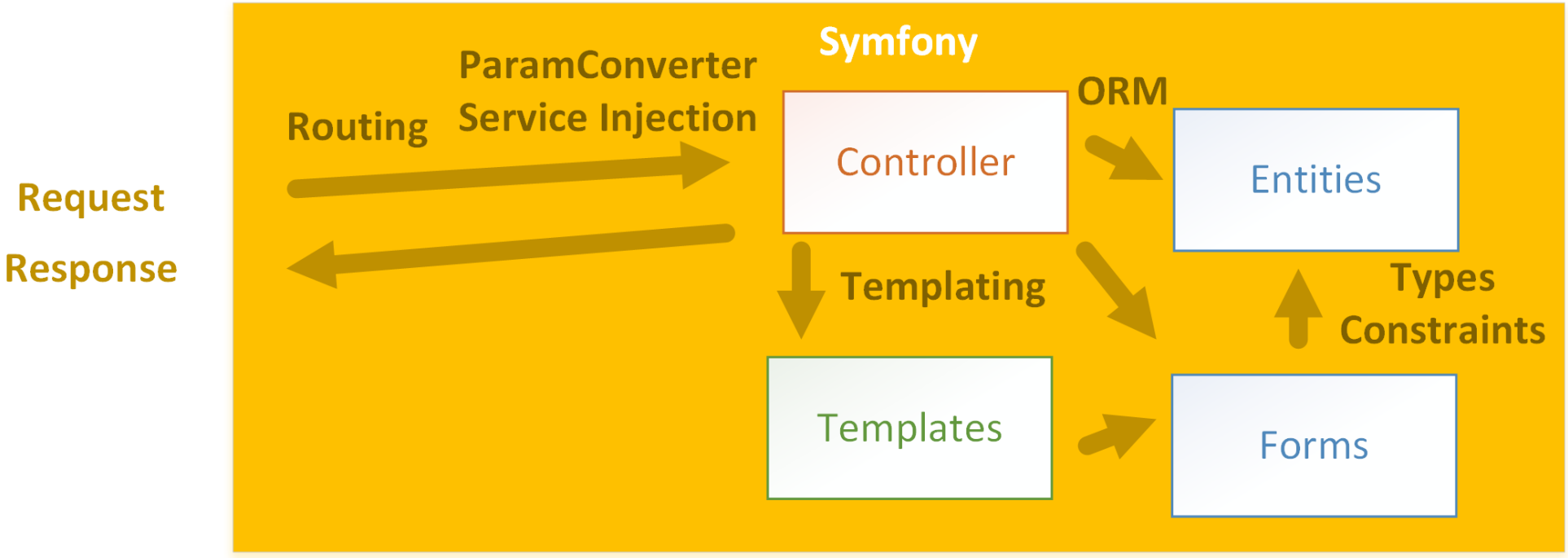
```
composer create-project symfony/skeleton my-symfony-project
cd my-symfony-project/
```

  - Weitere Pakete für Supero-Webseite auf Symfony-Basis:

```
composer require doctrine twig form validator security
composer require --dev profiler
composer require sensio/framework-extra-bundle
composer require doctrine/doctrine-fixtures-bundle
```

    - Symfony-Version von Supero auf GitHub:
      - https://github.com/fzieris/php-demo-supero-symfony
      - Die README.md-Datei ist randvoll mit mit Informationen

Zusammenfassung: Heutige Einheit



Danke!