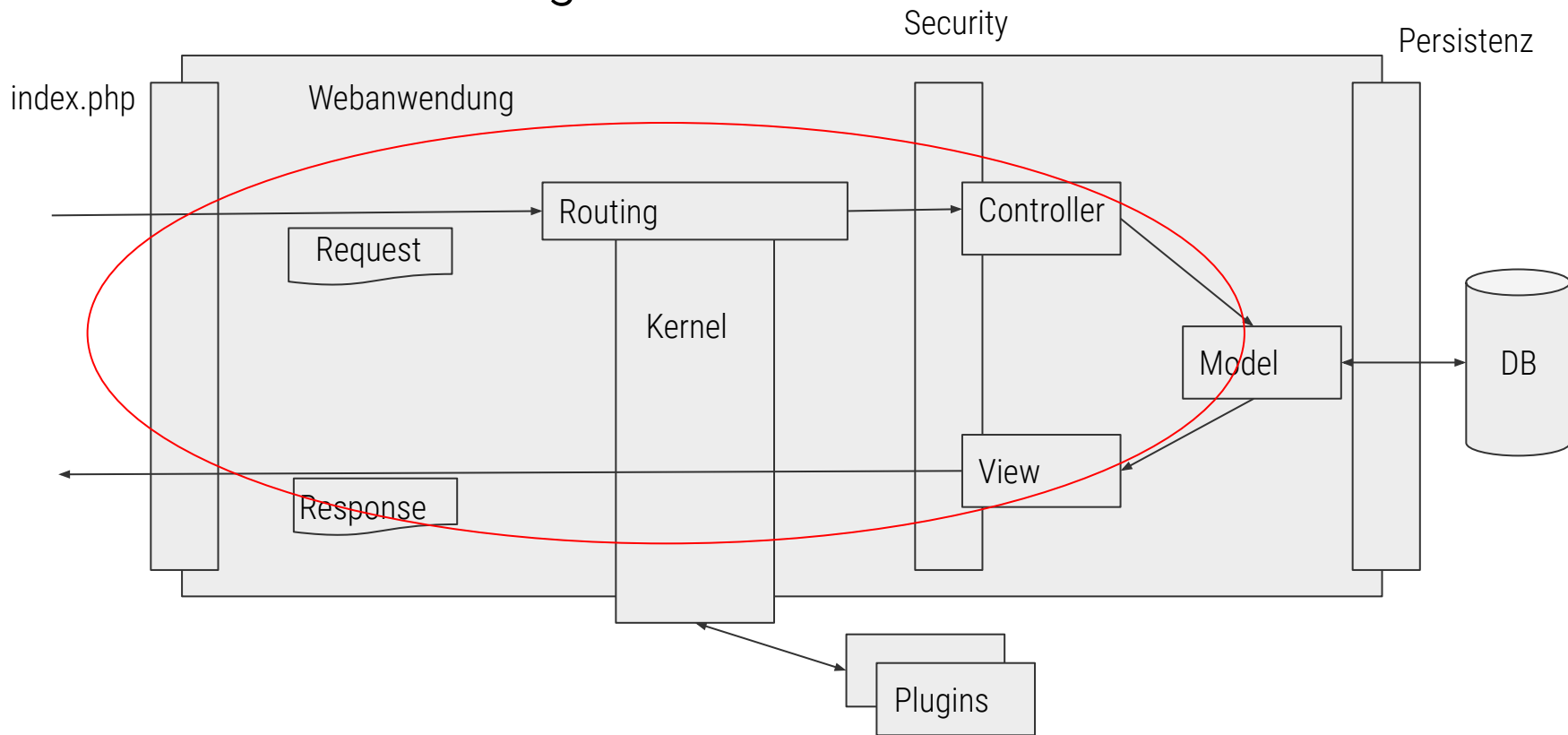


# Webentwicklung

FWPM

# Nutzerinteraktion über Controller und Views

# Aufbau Webanwendung

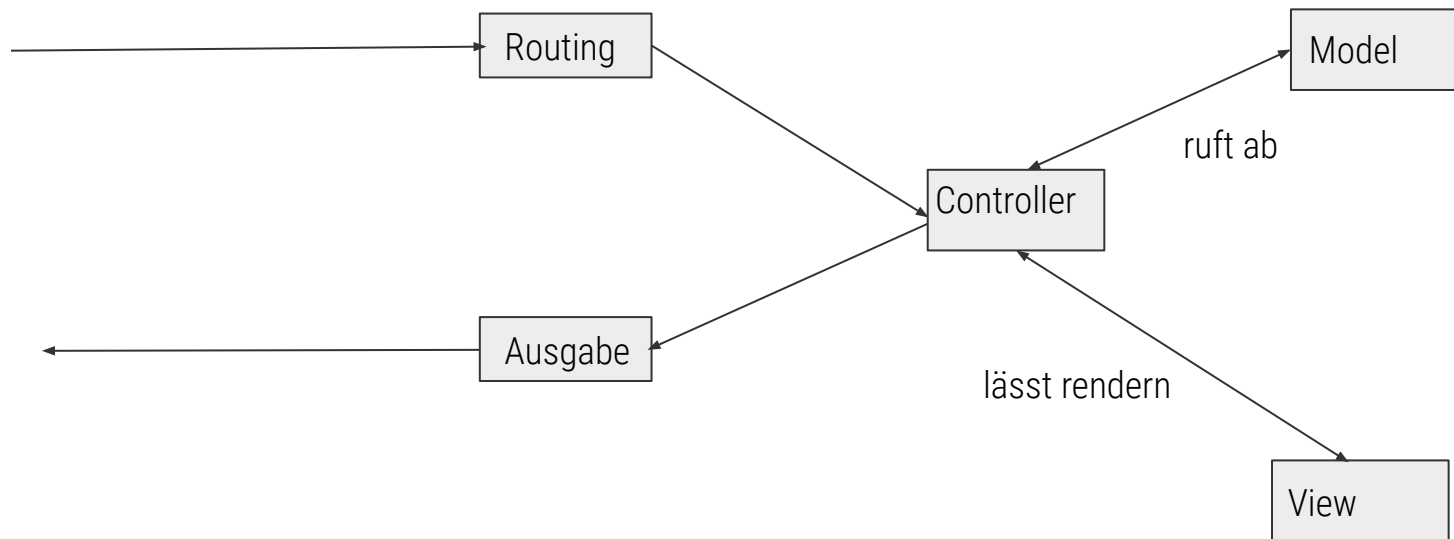


# Was sind Controller?

- Nimmt Nutzerinteraktion entgegen
  - Schnittstelle für den Nutzer
- Erster Feature-bezogener Einstiegspunkt in Anwendung
- Übernimmt:
  - Verarbeitung von Request
    - Z.B. Auswertung von Formulardaten
  - Zusammenfassen von Ausgabe
- Orchestriert Views für Rendering
  - Lädt Model und reicht an View weiter
- Wird klassisch durch Routing auf URL gemapped
- Business Logik im Controller? Jein!

```
class BlogController
{
    /**
     * @param RequestInterface $request
     * @param ResponseInterface $response
     */
    public function list(RequestInterface $request, ResponseInterface $response)
    {
        // load list of blog posts and render them using a view class
    }
}
```

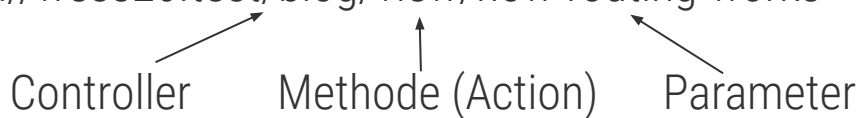
# Controller - Drehkreuz von Daten



# Routing zum Controller

- URLs repräsentieren Funktionalität unserer Anwendung
- Folgt oft festem Muster

Z.B. <https://wess20.test/blog/view/how-routing-works>



Controller      Methode (Action)      Parameter

- Aber wie andere Muster abbilden?
  - Registrieren von Routen
  - Mapping auf Controller/Action Kombination

# Routing zum Controller

- Mapping von Route auf Controller-Action sollte:
  - Leicht zu erweitern sein
  - Wenig geändert werden müssen (vergleiche Open-Closed)
- a) Dedizierte Konfigurationsdatei
  - Flexibel
  - Keine Code-Änderung nötig (aber Änderung an Datei)
- b) Controller kennt Route selbst
  - Etwas unflexibel
  - Perfektes Self-Containment
  - Oft über sog. Annotations
- c) Viel Pragmatismus dazwischen

```
# routes.yml
blog_list:
  path:      /blog
  controller: App\Controller\BlogController::list

blog_show:
  path:      /blog/show/{id}
  controller: App\Controller\BlogController::show
```

```
/**
 * @Route("/blog")
 */
class BlogController
{
    /**
     * @Route(path="list", methods="GET")
     *
     * @param RequestInterface $request
     * @param ResponseInterface $response
     */
    public function list(RequestInterface $request, ResponseInterface $response)
    {
        // load list of blog posts and render them using a view class
    }
}
```

# Routing zum Controller

- GET Parameter zum Routing?
- Vorteile:
  - Routing Aufwand verschwindend gering
  - Skaliert extrem einfach, da keine Pflege
    - URL-Kollisionen/Canonical URLs
    - Lokalisierung (I18N) der URLs
- Nachteile:
  - Schlecht für SEO
  - Für Menschen schwer lesbar
  - Faustregel: nur für interne Anwendungen sinnvoll



<https://learning-campus.th-rosenheim.de/course/view.php?id=3260#section-3>



# Router/Controller Kommunikation

- Brauchte alle Eingabedaten
  - Mehr oder weniger aufbereitet
  - Einfachste Lösung: unser komplettes Request Objekt
- Kümmert sich um Ausgabe
  - Response Objekt verfügbar machen

```
class BlogController
{
    /**
     * @param RequestInterface $request
     * @param ResponseInterface $response
     */
    public function list(RequestInterface $request, ResponseInterface $response)
    {
        // load list of blog posts and render them using a view class
    }
}
```

# Router/Controller Kommunikation

- Schönerer Weg: Parameter Parsing und Response als Rückgabe
  - `/blog/show/{id}`
  - Erlaubt feingranulare Controller Actions
    - Request nicht generisch im Controller ausgewertet
  - Aber: Wer übernimmt Parsing?
    - Oft zusätzliche Aufgabe des Routers
    - Benötigt gut abgestimmte Logik des Controller Aufrufs
      - Router darf kein Wissen über Controller haben!

```
class BlogController
{
    /**
     * @param string $headline
     */
    public function show(string $headline)
    {
        // load blog post by headline and render it
    }
}
```

```
$controller = new BlogController();
$controller->show( headline: 'how-routing-works');
```

# Router/Controller Kommunikation

- Wie wird Aufruf des Controllers umgesetzt?
- PHP kennt sog. *callable*
  - Pseudotyp um aufrufbaren Code programmatisch erkennbar zu machen
  - Vergleiche auch Callback Begriff
- Dynamische Aufrufe über Hilfsfunktionen
  - Oder über dynamisch benannte Klassen/Methoden

```
$callable = ['BlogController', 'show'];  
$response = call_user_func_array($callable, ['how-routing-works']);
```

```
$controllerName = 'BlogController';  
$methodName = 'list';  
  
$controller = new $controllerName;  
$response = $controller->$methodName($request);
```

# MVC - View

- Ist Repräsentation des Models
  - Je nach notwendiger Darstellung
    - Kümmert sich um Rendering in HTML
    - Konvertiert Daten zu JSON
    - ...
- Kann aus mehreren Einzelkomponenten bestehen
  - View Klasse
  - Render Engine
  - Template
  - ...

# Client-Side vs. Server-Side Rendering

- Wo wird DOM primär finalisiert?
  - Wo werden dynamische Inhalte in HTML umgewandelt
- Aktuelle Tendenz zu Client-Side Rendering
  - Über Javascript Frameworks wie React, Vue.js
  - Vorteil: Erzeugt Stateful Gefühl weil keine sichtbaren Requests
  - Daten zur Dynamisierung über Webservices
- Klassisch ist Web reines Server-Side Rendering
  - Inhalte werden über PHP zu HTML zusammengefasst
  - Komplettes HTML wird ausgeliefert
  - Kann architektonisch unterschiedlich realisiert werden

# View Klassen und Templates

- Über View Klassen werden Templates gerendert
- Über Zusammenspiel der View Klassen entsteht Mehrfachnutzung
  - Verhindert Wiederholung bei reinem Routen basiertem Rendering
- PHP als Skript unterstützt nativ, aber nicht zu empfehlen
- Lösung muss über abstraktes Templating her
  - Z.B. Template Engine
  - View Klasse für generelles Layout

```
<body>
  <h1>Welcome to my home page!</h1>
  <p>Some text.</p>
  <p>Some more text.</p>
  <?php include 'footer.php';?>
</body>
</html>
```

```
public function render($data)
{
    extract($data);
    ob_start();
    require dirname(dirname(__DIR__)) . '/view/templates/dynamic_content.html';
    return ob_get_clean();
}
```

```
<div class="item">
  <h3><?php echo $story->getTitle(); ?></h3>
  <p><?php echo $story->getContent(); ?></p>
</div>
```

# Komponenten/Komposition

- Erlauben die Wiederverwendung von Elementen der Seite
  - Z.B. Nutzerliste, Datepicker, Kontaktformular
- Bündelt View, Controller und Model
- Einhängepunkt über Nutzung im Template
  - Anstatt reiner Einstieg über Routen
  - Braucht Framework Unterstützung
- Status Quo für Client Rendering
  - Im Server Rendering seltener
- In Javascript als *Web Components* vom W3C spezifiziert

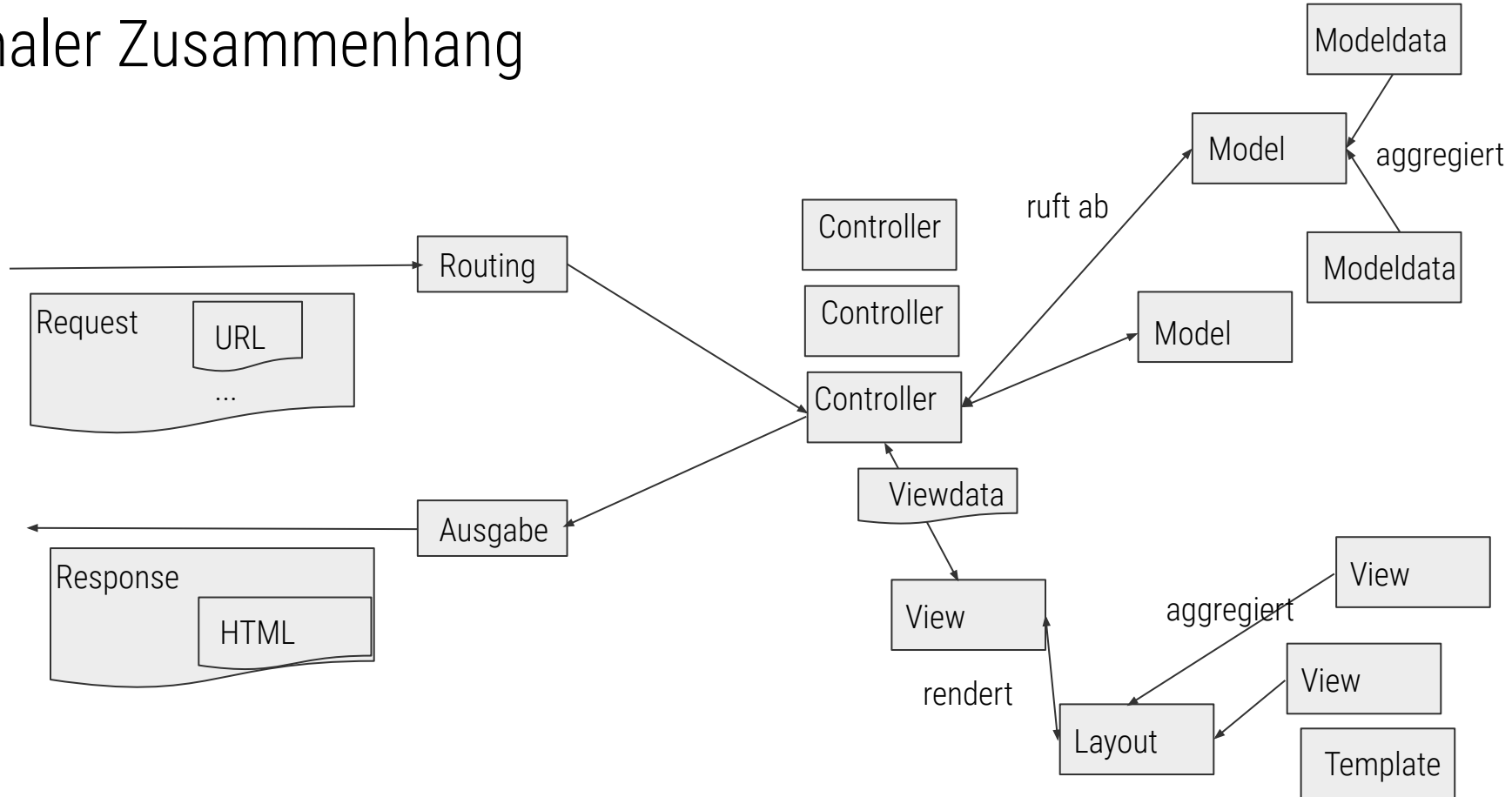
# Komponenten/Komposition

- Muss selbst programmiert werden
  - Z.B. Aufruf andere Controller und sammeln von gerenderten Templates
- Oft über zusätzliches **Layout** gelöst
  - Eine übergeordnete Viewklasse
  - Nutzt andere Views um Seitenstruktur zu rendern
  - Problem: Ansteuerung der Controller?
    - Nicht alle Controller sind über URL erreichbar

```
<div class="container">  
  <div class="header">{{header}}</div>  
  <div class="center">  
    <div class="sidebar">{{sidebar}}</div>  
    <div class="body">{{body}}</div>  
  </div>  
  <div class="footer">{{footer}}</div>  
</div>
```



# Finaler Zusammenhang



## Quellen:

- <https://blog.ircmacell.com/2014/11/a-beginners-guide-to-mvc-for-web.html>
- <https://blog.ircmacell.com/2014/11/alternatives-to-mvc.html>
- [https://de.wikipedia.org/wiki/Prinzipien\\_objektorientierten\\_Designs](https://de.wikipedia.org/wiki/Prinzipien_objektorientierten_Designs)



## Bildquellen:

- MVC Pattern By RegisFrey - Own work, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=10298177>
- Dependency Inversion Beispiel vermutlich von Sevenclev, Gemeinfrei, <https://commons.wikimedia.org/w/index.php?curid=1886669>

