

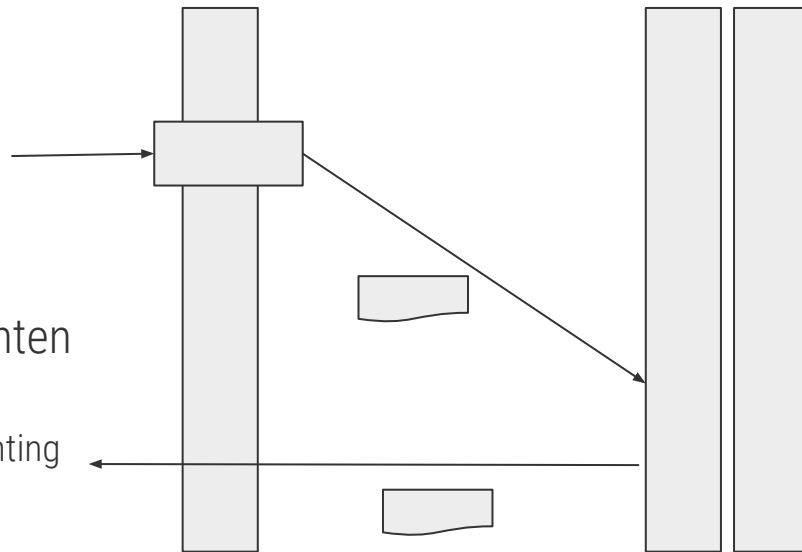
# Webentwicklung

FWPM

# Aufbau Webframeworks

# Bausteine eine (Web-) Anwendung

- Schichten zur Verarbeitung von Anfragen
  - Je nach Architektur
- Komponenten zum Ablaufkontrolle
  - Welche Teile der Anwendung sind benötigt
  - Verwaltung Ein- und Ausgabemechanismen
- Datenübertragung zwischen einzelnen Komponenten
  - Oft über Daten-Klassen sog **DTO**
  - Erlauben nachvollziehbare Übergänge z.B. durch Typehinting



# Request/Response - Kapselung

- Zur Datenübertragung
- Klasse um magische Konstanten wie `$_SERVER`, `$_F`
  - Kapselung nach Open-Closed Prinzip
- Kontrolle darüber WO Informationen verfügbar sind
  - Durch Weitergabe der Request Instanz
- Erlaubt zentrale Aufbereitung von Daten
  - Filterung, Validierung, Anpassung
- Ermöglicht Erweiterbarkeit um eigene Konzepte
  - Als Kindklasse von Request z.B. SoapRequest
- Eigener Standard: PSR-7

```
class Request
{
    protected array $parameters;

    protected array $context;

    public function __construct()
    {
        $this->context = $_SERVER;
        $this->parameters = $_REQUEST;
        // ... parse some more information here
    }

    public function hasHeader(string $headerName)
    {
        return isset(
            $this->context[$this->normalizeHeaderName($headerName)]
        );
    }

    public function getHeader(string $headerName)
    {
        return $this->context[$this->normalizeHeaderName($headerName)];
    }
}
```

# Request/Response - Ausgabe

- Kapselung der Antwort an den Client (Response)
  - Existiert sonst nur innerhalb der PHP Laufzeit
    - Wenig Kontrolle durch Entwickler
- Ebenfalls Vorteile bei Datenhandhabung
  - Z.B. auch zur Verarbeitung in mehreren Schritten
- Erlaubt zentrale Behandlung
  - Z.B. zentrales Handling von Status Codes
  - Handling von Media Types und anderen Headern

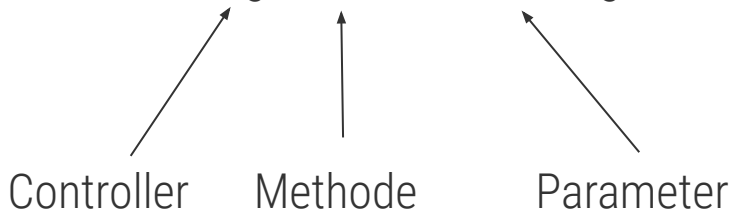
```
$request = Request::createFromGlobals();  
try {  
    $response = $kernel->handle($request);  
} catch (\Exception $e) {  
    $response->setStatusCode( code: 500);  
    $error = new GenericServerError($request);  
    $response->setContent($error->getMessage());  
}  
  
$response->send();  
$kernel->terminate($request, $response);
```

# Routing

- Zur Ablaufkontrolle
- Reicht Request Verarbeitung an richtiges Modul im Framework weiter
- Z.B. als Chain of Responsibility von spezialisierten Routern
- Verarbeitet Bestandteile der aufgerufenen URL
- Folgt oft festem Muster

Z.B. <https://we.test/blog/show/how-routing-works>

Controller      Methode      Parameter



```
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');
```

- Aber auch völlig freies Routing möglich z.B. <https://we.test/how-to-route-like-a-boss>

# Kernel

- Zur Ablaufkontrolle
- Nicht zwangsläufig notwendig
  - Verbessert aber Kapselung
- Koordiniert Request/Response Flow
- Organisiert Bootstrapping von anderen Komponenten
  - Z.B. Routing
- Hält zentrale Informationen
  - Z.B. Konfiguration
- Oft Einhängpunkt zur Registrierung von dynamischen Erweiterungen eines Frameworks
  - Plugins, Third-Party Module, neue View Templates, ...

```
class Kernel extends BaseKernel
{
    use MicroKernelTrait;

    private const CONFIG_EXTS = '{php,xml,yaml,yml}';

    public function registerBundles(): iterable{...}

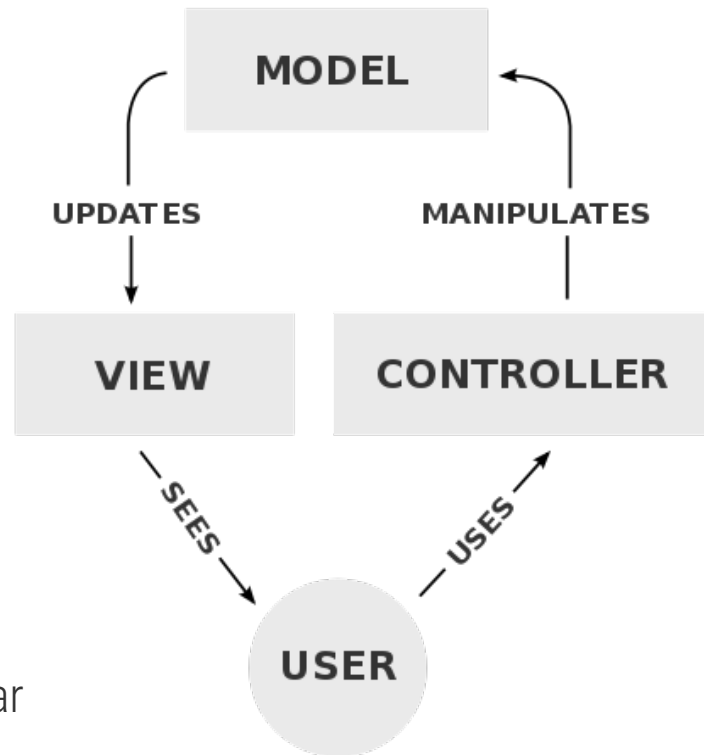
    public function getProjectDir(): string{...}

    protected function configureContainer(ContainerBuilder $container)
    {
    }

    protected function configureRoutes(RouteCollectionBuilder $routes)
    {
    }
}
```

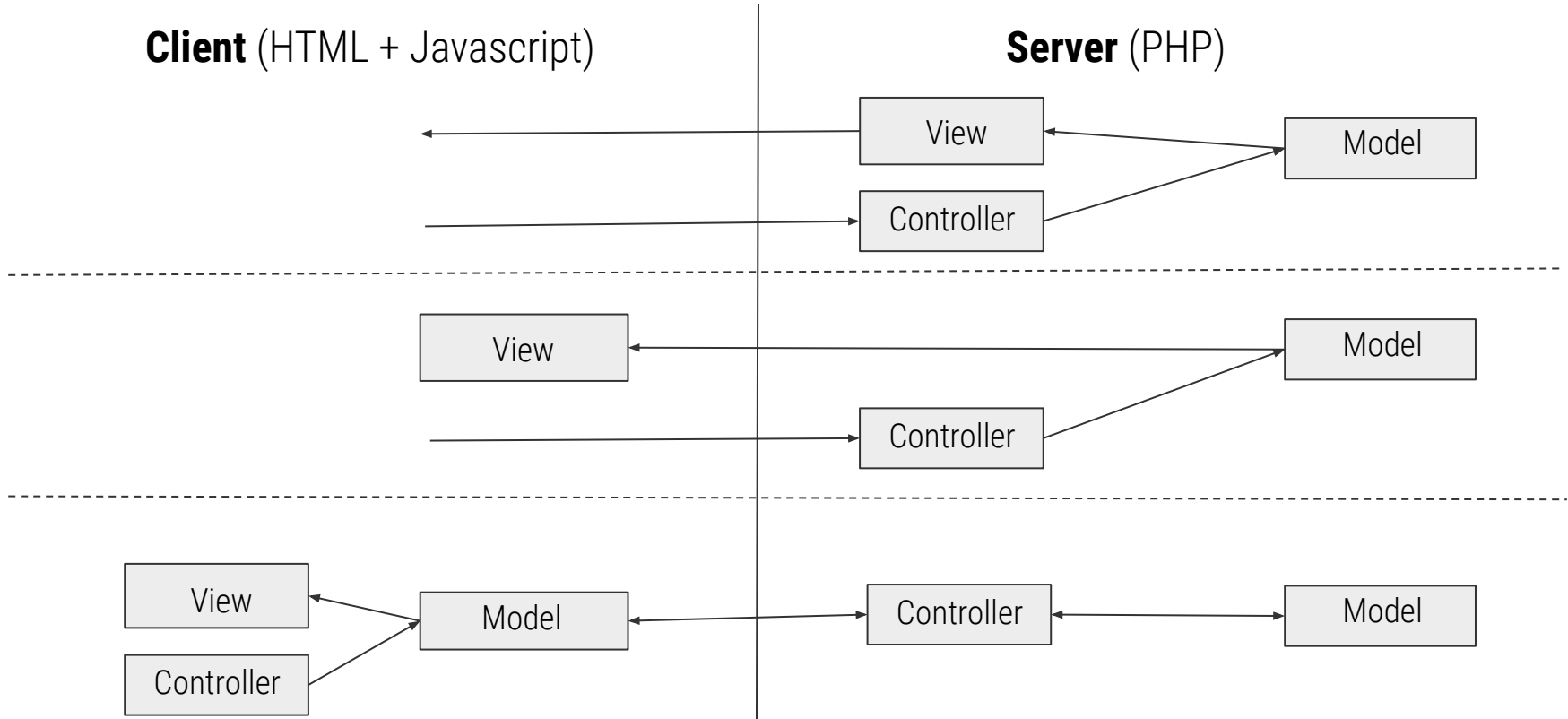
# MVC

- Verarbeitende Schichten
- **M**odel **V**iew **C**ontroller (1970er)
- Konzept zur Umsetzung von Separation of Concerns
- Vergleiche **Schichtenarchitektur**
- Trennt:
  - **Darstellung**
  - **Verwaltung der Daten**
  - **Nutzerinteraktion**
- Im Original aus der GUI Entwicklung
- Klassisch zu 100% auf dem Server
- Durch JS Frameworks auch auf Client/Server verteilbar
  - Z.B. View innerhalb einer SPA





# MVC und das Web - Thin Client/Fat Client



# MVC und das Web - Pattern

- **Problem:** MVC ist ein stateful Pattern (Model/View Beziehung)
- Das klassische Web ist aber im Kern weiterhin stateless
- Kein Framework nutzt MVC zu 100%
  - Jeweils unterschiedliche Auslegungen
    - Z.B. Controller als Koordinator zwischen Model und View
    - View die sich Model selbst lädt
    - ...
  - Lange Historie und top Separation of Concerns sorgen für Verbreitung
- Deshalb: **Nicht zu religiös werden**
  - **Hauptsache: Separation of Concerns**

# MVC - Controller

- Nimmt Nutzerinteraktion entgegen
  - Schnittstelle für den Nutzer
- Manipuliert Model (Schreibzugriff)
- Klassisch nicht mehr!
- Oft Interaktion mit View
  - Lädt Model und reicht an View weiter
- Enthält/ist Schnittstelle zu Business Logik

# MVC - Model

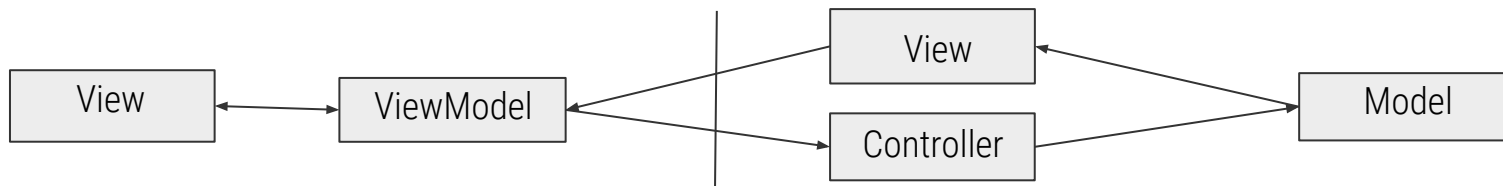
- Beinhaltet/Ist Daten der Anwendung
- Oft direkte Entitäts-Klassen z.B. "User" oder "BankAccount"
  - Ein Property pro Datenbankspalte
  - Getter/Setter
- Können Logik zur Abfrage aus DB enthalten
  - Oft aber nur rein zum Datentransport
- Pflege über Controller
- Falls möglich (stateful) Kommunikation mit View bei Änderungen

# MVC - View

- Ist Repräsentation des Models
  - Je nach notwendiger Darstellung
    - Kümmert sich um Rendering in HTML
    - Konvertiert Daten zu JSON
    - ...
- Updated sich teilweise selbst (lädt Model)
  - Wird teilweise von außen (Model oder Controller) aktualisiert
- Am Einfachsten auf Client Seite zu verlagern

# Alternative Architekturen - MVVM

- **M**odel **V**iew **V**iew**M**odel (2005)
- Erleichtert Verteilbarkeit von MVC
- Erlaubt eine Stateful View/Model Kombination => das ViewModel
  - Kombiniert Vorteile von Stateful und Stateless
  - Nutzt sog. Binder zum State Management zwischen View und ViewModel
- Ansatz moderner JS Frameworks
  - Vue.js
  - Angular



# Cross Cutting Concerns

- Belange die sich nicht kapseln lassen
  - “Schneiden” quer durch andere Schichten und Module
- Haben oft Hilfsfunktionen
  - Aber fundamental wichtig
- Z.B. Logging, Security, Lokalisierung, auch DB Zugriff ...
- Separation of Concerns ist schwer einzuhalten
  - Kapselung würde Nutzung erschweren
  - Mehrfachimplementierung sorgt für Redundanz und ist fehleranfällig
- Keine 100% Lösung möglich

# Cross Cutting Concerns - Integration

- Integration immer von Anwendungszweck abhängig
  - Factory, Dependency Injection, Aspekt-orientierte Programmierung

- Relativ sinnvolles Vorgehen:

- Kapselung in Design Pattern
- Nutzung als Factory
  - Für Infrastruktur wie Logging
- Nutzung per Dependency Injection
  - Für Essentielles wie Security, DB, ...

- Königsweg: AOP

- Extrem mächtig
- Sehr aufwändig im Framework

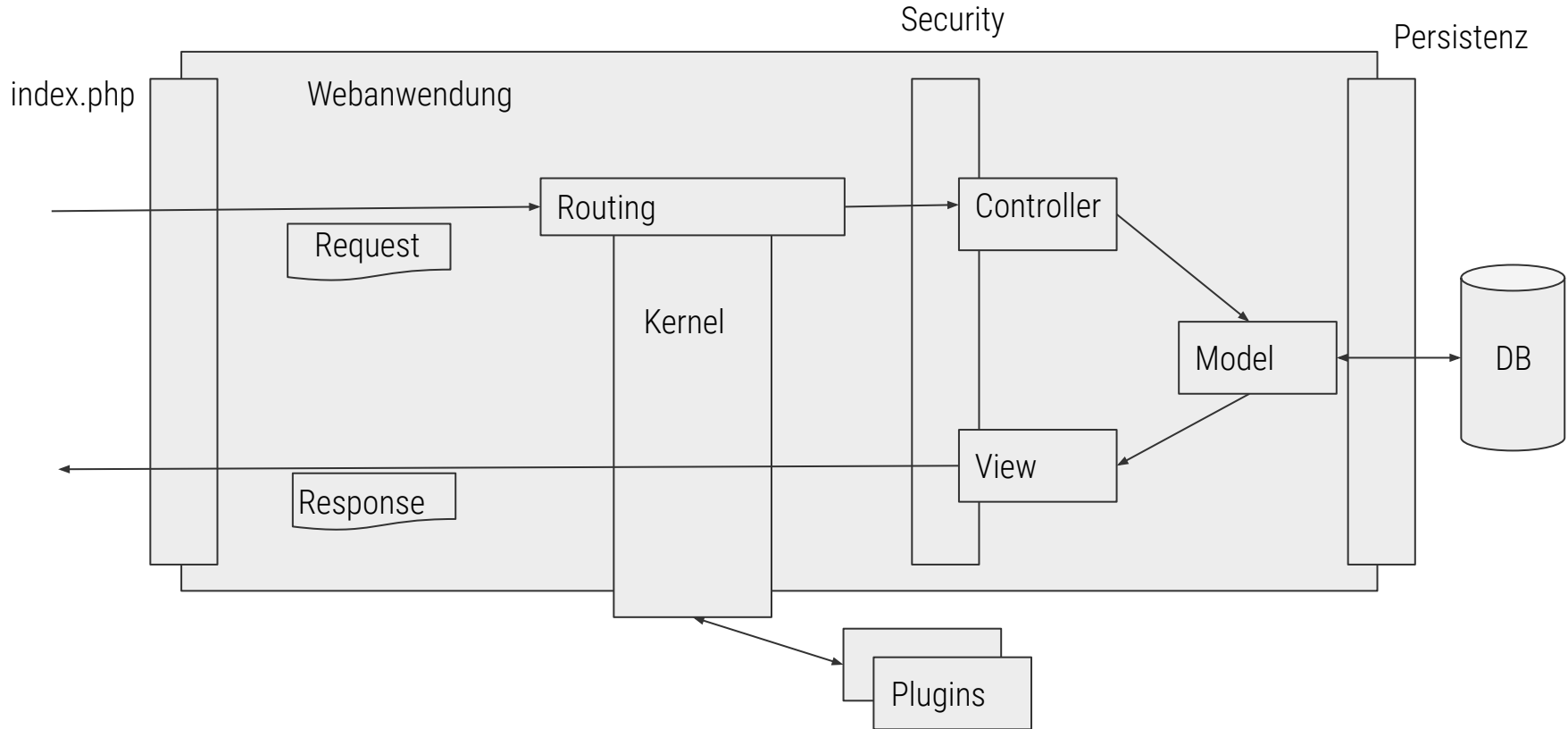
```
class CatalogController
{
    protected RepositoryInterface $productRepository;
    protected ViewInterface $view;

    public function __construct(RepositoryInterface $productRepository)
    {
        $this->productRepository = $productRepository;
    }

    public function show()
    {
        $products = $this->productRepository->findAll();
        $this->view->render($products);
    }
}
```



# Aufbau Übersicht



## 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>

