Webservices

Was sind Webservices?

- Anwendungsschnittstelle zur **maschinenlesbaren** Kommunikation
  - Kommunikation von Anwendungen untereinander
  - Webanwendungen sind für Menschen optimiert, daher keine Webservices
- Liefern Daten über Netzwerkprotokolle aus
  - Nicht zwingend über das Internet
- Erlauben Steuerung von Anwendung über Netzwerk
- Nutzen Datenaustauschprotokolle
  - Z.B. XML, JSON, protobuf

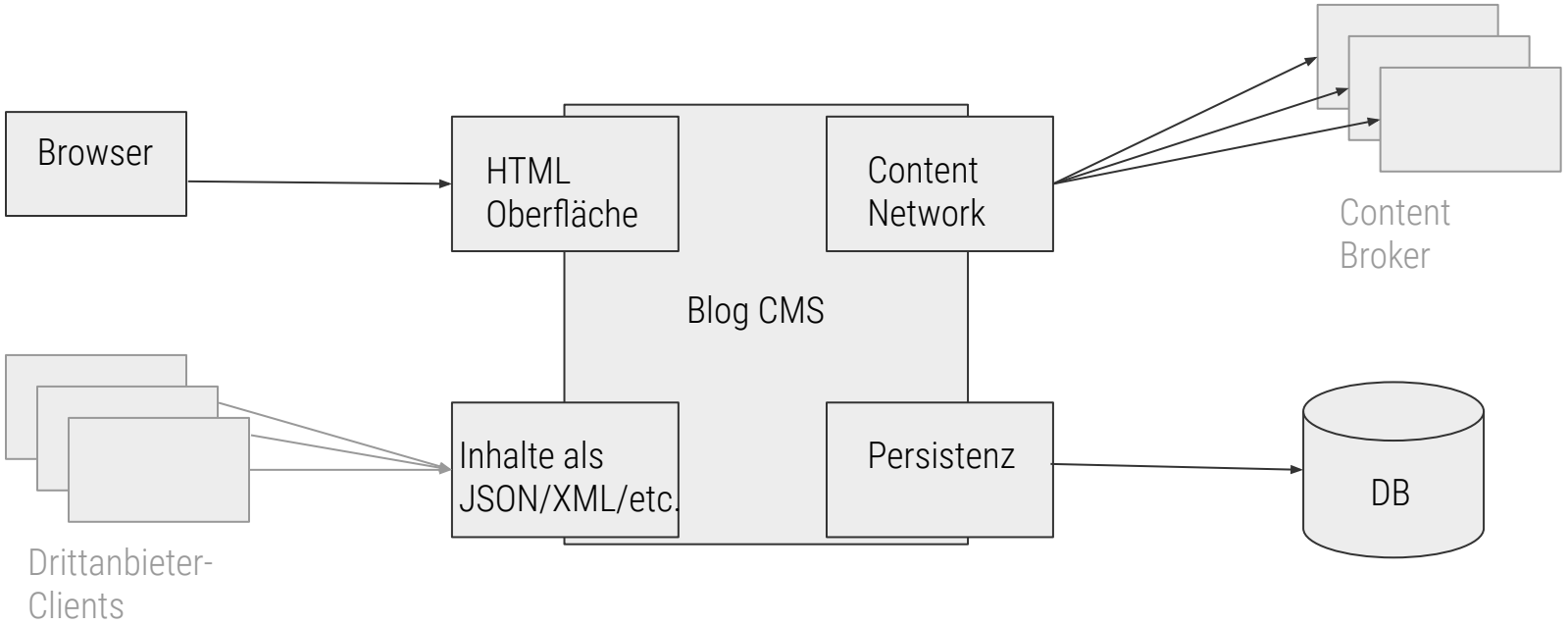
Warum Webservices?

- Erlauben das Anbinden von anderen Anwendungen
  - Zur Nutzung derer Daten
    - Erlaubt “Normalisierung” auf Service-Ebene
- Verringert Kopplung von Software-Komponenten
  - Ermöglicht Austausch und bessere Wartung
- Zur Nutzung unterschiedlicher Technologien in einem Projekt
- Komponenten können unabhängig voneinander entwickelt werden
  - Unterschiedliche Teams, unterschiedliche Firmen, etc.
- Skalierbarkeit der Anwendung wird erhöht

Warum nicht?

- Erhöhen Komplexität einer Anwendung
  - Verteilte Verarbeitung ist schwer zu verstehen
  - Pflege mehrere Komponenten in eigenen Repositories
- Verringern Performance durch Netzwerkübertragung
- Machen abhängig von Drittparteien
  - Z.B. Datenhaltung durch andere Firma, Nutzung von Google Diensten
- Bieten neue Angriffsvektoren durch Netzwerkanbindung

Warum Webservices? - Usecase Blog CMS



Arten - Middlewarebasiert

- Bieten zentralen Punkt für Aufrufe
  - Z.B. URL https://soap.webentwicklung.test
  - Alle Anfragen gehen an diese Adresse
- Nachrichten müssen gewünschte Aktion beinhalten
  - Z.B. Abfrage von BlogPosts der letzten 3 Tage
- Bieten Funktion zur Beschreibung der verfügbaren Dienste
  - Z.B. WSDL bei SOAP
- Vergleich: Hauspost. Nachrichten erreichen Empfänger nicht direkt

Wichtige Vertreter - SOAP

- W3C Empfehlung (2003)
  - Vielfach genutzte Technik (vor allem Microsoft Welt)
- Nutzt XML als Übertragungssprache
- Überträgt Objekte, Befehle und Dateien
- Sehr flexibel in Nutzung
  - Durch Schema frei definierbarer Body
  - Verschiedene Protokolle
- Relativ schwergewichtig durch Metadaten
  - Besteht aus Envelope, Header und Body
- Nutzt Definition über WSDL

```xml
<?xml version="1.0"?>
<s:Envelope xmlns:s="https://www.w3.org/2003/05/soap-envelope">
    <s:Body>
        <m:TitleInDatabase xmlns:m="https://soap.webentwicklung.test">
            DOM, SAX und SOAP
        </m:TitleInDatabase>
    </s:Body>
</s:Envelope>
```

Wichtige Vertreter - GraphQL

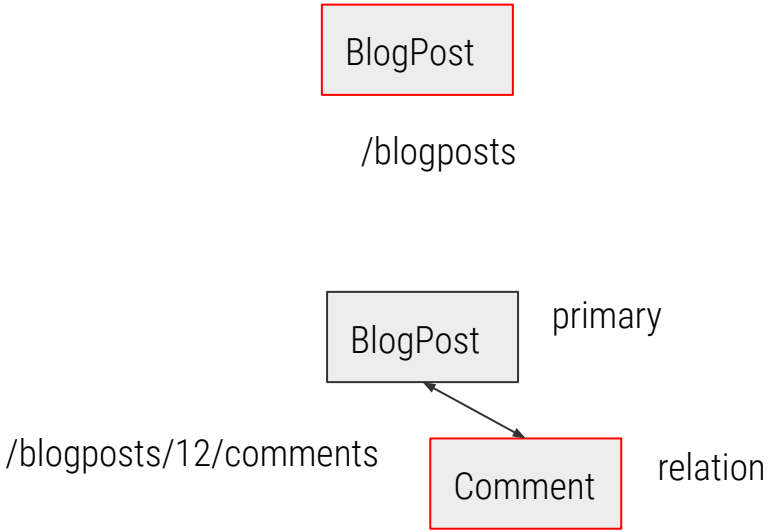
- Datenabfragesprache (2015 durch Facebook)
  - Kann auch schreiben (sog. Mutations)
  - Sehr populär
- Erlaubt extrem spezifische Abfragen
  - Verhindert unnötige Datenübertragung
- Benötigt volles Wissen über Datenstruktur
- Schema wird in SDL zentral dokumentiert und veröffentlicht
- Erlaubt Nutzung von Variablen und Methoden in Queries

```
{
    hero {
        name
        appearsIn
    }
}
```

>> https://graphql.org/learn/queries/

Arten - Ressourcenbasiert

- Mehrere Endpunkte
  - Gruppiert nach jeweiliger Ressource (vergleiche Model)
  - Folgen klarem URL Schema
  - Schema entspricht (öffentlicher) Datenstruktur
- Populär durch REST
- Kann Relationen in URL abbilden
- Extrem flexibel
  - Alles was als URL abbildbar ist
- Sehr leichtgewichtig
- Befehle über Ressourcen abbildbar



Wichtige Vertreter - REST

- **Re**presentational **S**tate **T**ransfer (2000)
- Keine Spezifizierung/Standard, sondern Architekturstil
  - Es gibt keine Standardimplementierung
  - In der Praxis oft als RESTful zu finden
- Rein Ressourcenbasierter Ansatz
- Oft für CRUD Anwendungen genutzt
  - **C**reate, **R**ead, **U**pdate, **D**elete
- Über HATEOS auch Hyperlinkfähig
  - Antwort vom Server enthält weiterführende Links
- Nutzt sog. Repräsentationen einer Ressource
  - Format der Daten, oft JSON

Wichtige Vertreter - REST

- Kombination aus URL und HTTP Methode bildet Semantik
- **GET** - Abrufen von Ressourcen
- **POST** - Erzeugen von Ressourcen
- **PUT** - Ändern von bestehenden Ressourcen
- **DELETE** - Löschen von Ressourcen

GET /blogposts - Abrufen aller Blog Posts
DELETE /blogposts/13 - Löschen des Blog Posts mit der ID 13
PUT /blogposts/simple-rest-interface - Ändern eines bestimmten Blog Posts
POST /blogposts/12/comments - Erstellen eines Kommentars zu Blog Post mit der ID 12

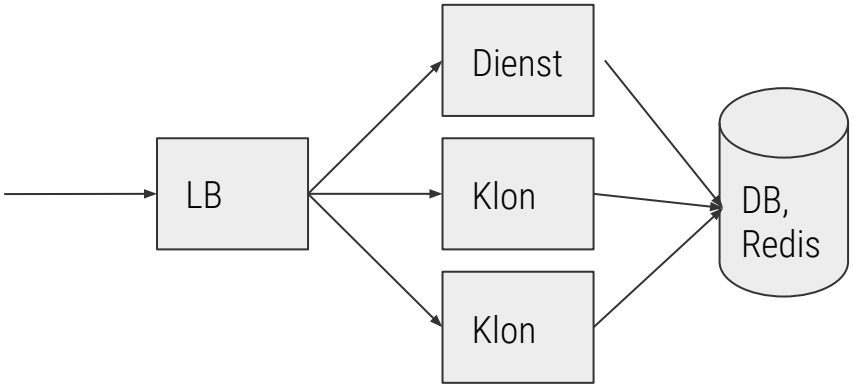
>> https://petstore.swagger.io/

Vorteile - Validierung

- Maschinenlesbarkeit meint auch **Prüfbarkeit durch Maschinen**
  - Automatische Validierung relativ einfach
- Webservice Konzepte nutzen Schemas
  - Teilweise fest eingebaut (SOAP, GraphQL)
- Validierung notwendig, da (Nutzer-)Eingabe!
- JSON Validierung über JSON Schema Spezifikation
  - PHP Bibliotheken z.B. justinrainbow/json-schema
- XML hat native Schema Funktionalität

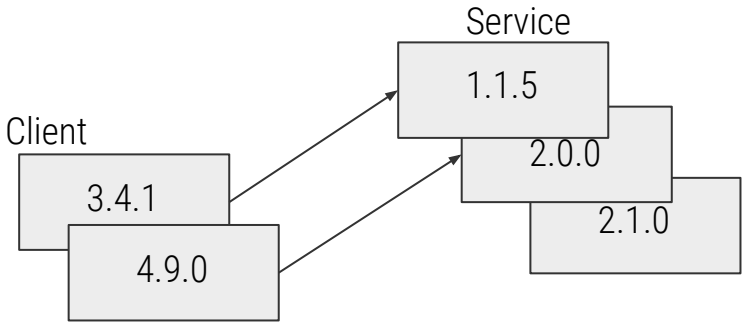
Vorteile - Skalierbarkeit

- Lose gekoppelte Systeme lassen sich besser skalieren
  - Weniger Rücksicht auf Nutzung nötig
- Horizontale Skalierung durch Duplizieren eines Services
  - Eventuell über Load Balancer verteilt
- Vertikale Skalierung durch Auslagern von Diensten auf eigene Hardware
- Globaler State (z.B. Sessions) schwierig
  - Braucht oft extra Dienst z.B. Redis



Vorteile - Wartbarkeit

- Lose gekoppelte Systeme lassen sich besser warten
  - Komponenten können leichter ersetzt werden
  - Failover Szenarien sind leicht machbar
- Webservices werden einzeln versioniert
  - Angebot von mehreren Versionen sinnvoll
    - Migration nach Lebenszyklus
- Aber: Webservice(s/-anbindungen) sind aufwändig zu testen
  - Oft Mocks für lokale Entwicklung nötig

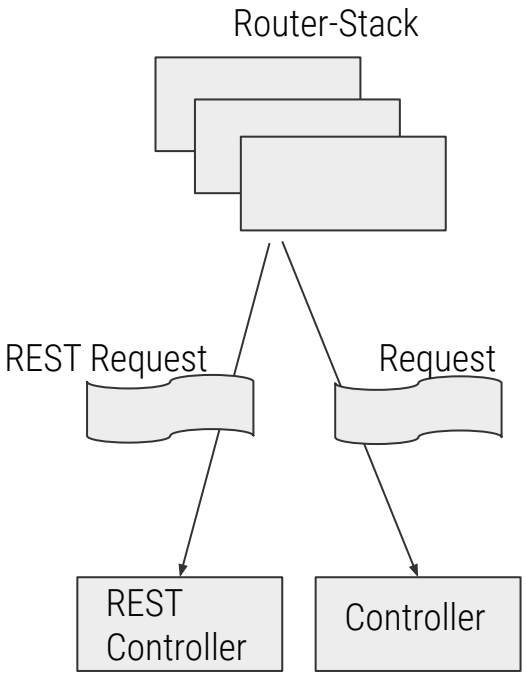


REST in der Praxis - Überblick

1. Neues Routing
   a. REST URL Struktur muss erkennbar und verarbeitbar sein
2. Übergabemechanismus an Controller schaffen
   a. Dynamische URL Inhalte an Controller übergeben
3. Neue Controller Struktur
   a. Muss Daten anders aufbereiten
   b. Muss andere Views verwenden
4. REST optimierte Views verwenden
   a. Serialisierung von Daten im Mittelpunkt

REST in der Praxis - Neues Routing

- URLs folgen jetzt fixem Schema
  - Lassen sich auf Klassenstruktur abbilden
  - Beinhalten dynamische Elemente
- HTTP Methode muss berücksichtigt werden
  - Hat explizit Einfluss auf Verarbeitung
- Neues Routing notwendig
- Dynamische Teile müssen Controller erreichen
  - Braucht Identifier zur Verarbeitung



REST in der Praxis - Neues Routing

```php
$registrationRouter = new RegistrationRouter($request);
$registrationRouter->addRoute( route: '/', controllerName: BlogPost\Overview::class);
$registrationRouter->addRoute( route: '/blogpost/add', controllerName: BlogPost\Add::class);

$restRegistrationRouter = new RestRegistrationRouter($request);
$registrationRouter->addRoute( route: '/blogposts/{id}', controllerName: Rest\BlogPosts::class);
$registrationRouter->addRoute( route: '/comments/{id}', controllerName: Rest\Comments::class);
$registrationRouter->addRoute( route: '/blogposts/{id}/comments/{id}', controllerName: Rest\Comments::class);

$restConventionRouter = new RestConventionRouter($request);

$fallbackRouter = new FallbackRouter($request);

$routers = [
    $registrationRouter,
    $restRegistrationRouter,
    $restConventionRouter,
    $fallbackRouter,
];
```

REST in der Praxis - Übergabemechanismus

- Einfache Übergabe ist immer möglich
  - Identifier direkt an Controller
  - Assoziatives Array bei verschachtelter API
- Besser: Request DTO erweitern
  - *RestRequest* der von *Request* erbt enthält Identifier
  - Erlaubt einheitliche Controller Schnittstelle *execute()*

```
$controller->execute($request, $response);
```



```
$controller->execute($request, $identifier, $response);
```

```php
class RestRequest extends Request
{
    /**
     * Contains the REST identifiers from the URL.
     * E.g. /blogposts/12/comments would be ['blogposts' => 12, 'comments' => null]
     *
     * @var array $restIdentifiers
     */
    protected $restIdentifiers = [];

    /**
     * Contains the HTTP method of the request e.g. POST
     *
     * @var string $method
     */
    protected $method;
```

REST in der Praxis - Controller

- Wenig Änderung notwendig durch MVC Modell
- Müssen HTTP Methode unterscheiden können
- Je nach Methode andere Handhabung
  - Auslagerung in einzelne Controller möglich (*BlogPostsGet*, *BlogPostsPost*, …)
  - Ansonsten enge Kopplung mit Routing
    - Aufruf von *execute()* nicht mehr ausreichend
- Muss Views für REST Repräsentationen nutzen

```php
public function execute(Request $request, Response $response)
{
    $repository = new BlogPostRepository();
    $blogPosts = $repository->get();

    $view = new View\Json();
    $response->setBody($view->render($blogPosts));
}
```

REST in der Praxis - Views ersetzen

- Darstellung in HTML nicht zielführend
- Daten liegen bereits als Model-Klassen vor
- Serialisierung in Zielformat nötig
  - Überführung von Objektinstanz in String Repräsentation
  - Viele Standardbibliotheken vorhanden
- JSON Serialisierung nativ über *json_encode()*
- XML über *XMLWriter*, *SimpleXML*
  - Oder PHP Erweiterungen wie *XML_Serializer*
- View serialisiert statt Template Nutzung

```php
namespace WickEd\WESS20\View;

/** Class Json ...*/
class Json implements ViewInterface
{
    /**
     * @inheritDoc
     */
    public function render($data): string
    {
        return json_encode($data);
    }
}
```

REST in der Praxis - JSON Serialisierung

```php
class BlogPost
{
    /**
     * @var int $id
     */
    public int $id;

    /**
     * @var string $title
     */
    public string $title;

    /**
     * @var string $text
     */
    public string $text;

    /**
     * @var string $author
     */
    public string $author;
}
```

```php
$blogPost = new BlogPost();
$blogPost->id = '1';
$blogPost->author = 'Bernhard Wick';
$blogPost->title = 'JSON serialize';
$blogPost->text = 'That stuff is easy';

$result = json_encode($blogPost);
```

```json
{
  "id": "1",
  "author": "Bernhard Wick",
  "title": "JSON serialize",
  "text": "That stuff is easy"
}
```

REST in der Praxis - JSON Serialisierung

- Prinzip des Information Hidings
  - protected/private Property + Getter/Setter
- *\JsonSerializable* Interface hilft
- Erlaubt Vorbereitung zu serialisierender Daten
  - Erlaubt Bereinigung und Filterung
  - Erlaubt Handhabe für Abhängigkeiten
- Danach Nutzung mit *json_encode()* möglich

```php
class BlogPost implements \JsonSerializable
{
    /**
     * @var string
     *
     * @ORM\Id
     * @ORM\Column(type="string")
     * @ORM\GeneratedValue(strategy="UUID")
     */
    protected $id;

    // ...

    /**
     * @return mixed|\stdClass
     */
    public function jsonSerialize()
    {
        $result = new \stdClass();
        $result->author = $this->getAuthor();
        $result->title = $this->getTitle();
        $result->text = $this->getText();
        return $result;
    }
```

Quellen:

Bildquellen:

-