



Hochschule
Albstadt-Sigmaringen

Albstadt-Sigmaringen University

Webframeworks 1x1



Dipl. Ing. Sven Eppler (FH)
sodge IT GmbH



**Was tut das Webframework
für mich?**

Conv-over-Conf vs. Conf-over-Conv

- **Convention-Over-Configuration**
 - Code First, think later
 - Die Konvention definiert die Projektstruktur/Methodik
 - Defaults nehmen einem Entscheidungen ab
 - "Man kommt schnell vom Boden weg"
 - Kennt man die Konvention nicht, ist man "verloren"
 - Weicht man vom HappyPath ab, ist die Konvention oft im Weg

Conv-over-Conf vs. Conf-over-Conv

- **Configuration-Over-Convention**
 - Think first, code later
 - Funktionalität ist erst nach der Konfiguration verfügbar
 - Deutlich längere "Bootup"-Phase
 - Passt sich besser dem Problem an
 - Erfordert mehr Erfahrung/vorausschauende Entscheidungen
 - In-Depth-Verständnis des Gesamtsystems

HTTP Request Parsing

- Der HTTP-Request schlägt als erstes am Framework auf
- Zerlegen des Requests
 - Request-Line
 - Header
 - Body
- Bereitstellen eines "Requests"-Objektes für später
- Weitergabe des Requests an einen Dispatcher/Router

Dispatcher // Router

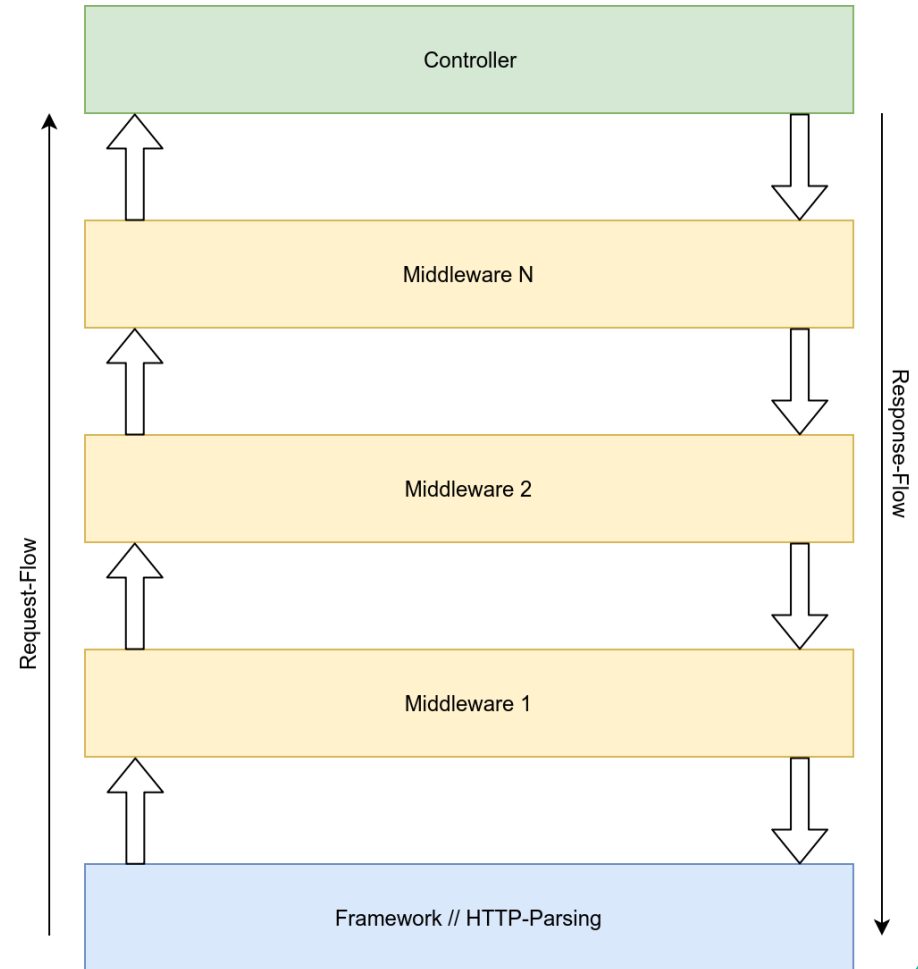
- **Match den HTTP-Request auf eine Action**
 - Action = Der Quellcode
- **Mögliche Regeln**
 - HTTP-Verb
 - Request-Header
 - Request-URL
 - Authentifizierter-Benutzer
- **Mögliche Ziele**
 - Actions in Controllern
 - Selten: Lambdas (In-Line-Actions)

Controller

- Eine Klasse die meist auf "Controller" endet
- Erbt von einer "Controller"-Klasse des Frameworks
- Ein Controller implementiert Actions
- Actions sind das Ziel von Routen
- Im Controller wird Ihre "Business Logic" aufgerufen
- Der Controller ist Vermittler zwischen dem HTTP-Request und Ihrem Code

Middlewares

- Middlewares sind Schichten zwischen HTTP-Parsing und Controllern
- Können den Request vorzeitig beenden (z.B. Auth)
- Können die Response modifizieren



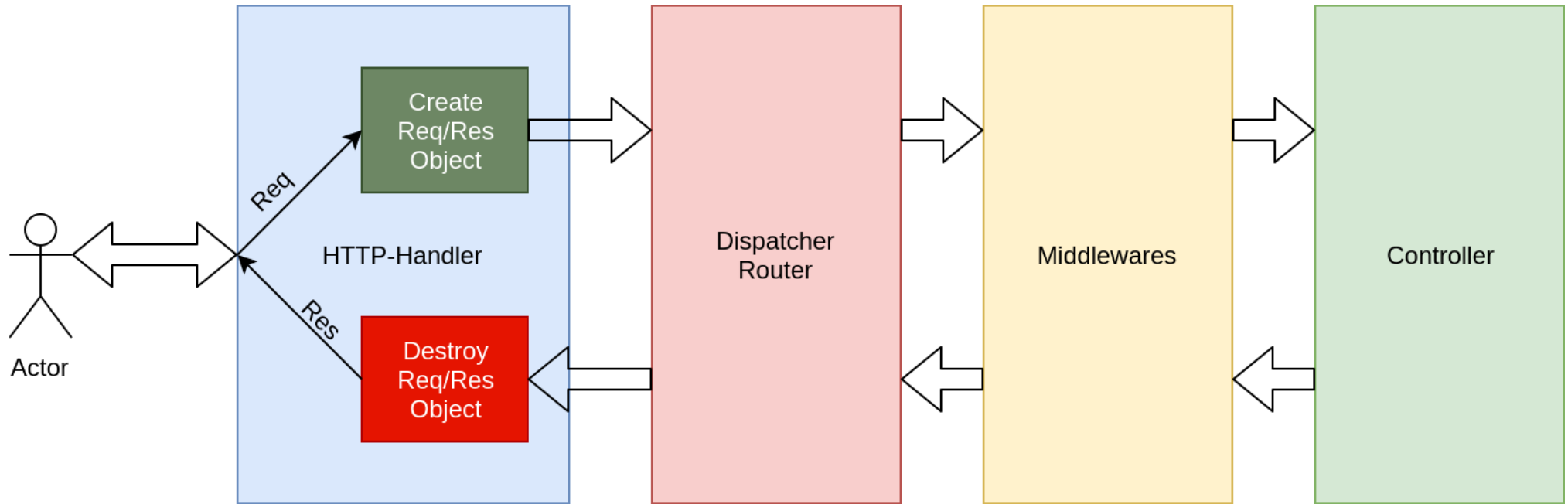
Response generieren

- **Das Framework erzeugt ein Response-Objekt**
 - Setzen des HTTP-Status Code
 - Setzen des HTTP-Headern
 - Setzen des HTTP-Body
- **Middlewares übernehmen die Konvertierung von "High Level Objects"**
 - JSON Formatierung, Bilder ausgeben, Template-Engine, etc.
- **Low-Level-Response-Manipulation selten notwendig**
- **Das Response-Objekt wandert wieder durch die Middleware zurück**

Request-Lifetime

- Request-Lifetime meint:
 - Wie lange "leben" Request/Response spezifische Objekte innerhalb des Frameworks
- Jeder Request ist isoliert, hat also seinen eigenen Context (z.B. eigene Request/Response Objekte)
- Sobald die Response auf die Leitung geschrieben wurde, wird der Context zerstört und die Request-Lifetime endet
- Im Request/Response Objekt können keine Daten persistiert werden!
- Daraus ergibt sich: Ein Request ist stateless!

Request Lifetime



Model-View-Controller (MVC)

- Pattern für Software-Design
- Trennung von:
 - Datenhaltung (Model)
 - Anzeige (View)
 - Geschäftslogik (Controller)
- Auch nicht-MVC-Frameworks implementieren Teile davon (z.B. Controller)
- Frameworks:
 - Ruby on Rails, Mojolicous, Spring MVC, Django, ZendFramework

A white speech bubble with a dark blue background. The bubble has a rectangular body and a triangular tail pointing downwards and to the left. The text is centered within the rectangular body.

**Wie sieht das in der Praxis
aus?**

Route Configuration in Express

- Die „App“-Instanz von Express liefert Basis-Routing:
 - `app.get("/", (req, res) => { ... });`
 - `app.post("/avatar/upload", (req, res) => { ... });`
- Das „Router“-Objekt abstrahiert das Routing von der App, dadurch ist es austauschbar/übertragbar
 - `var router = express.Router();`
 - `router.get("/", (req, res) => { ... });`
 - `router.post("/avatar/upload", (req, res) => { ... });`
 - `app.use("/", router);`
- Details:
<https://expressjs.com/en/guide/routing.html>

Basis Routing mit Express

```
var express = require('express');  
var app = express();  
  
app.get("/", (req, res) => {  
  res.send("Hello World!");  
});  
  
app.post("/path/to", (req, res) => { ... });  
  
module.exports = app;
```

Routing mit Router in Express

```
var express = require('express');
var router = express.Router();
var app = express();

router.get("/", (req, res) => {
  res.send("Hello World!");
});

router.post("/path/to", (req, res) => { ... })

app.use('/', router);

module.exports = app;
```


Basis Routing vs. Router Instanz

- **Basis Routing**

- Sofort vorhanden
- Routing sind fest mit der App verdrahtet
- Für kleine Anwendungen übersichtlich
- Komplexe Routing-Strukturen machen die App unübersichtlich und schlecht wartbar

- **Router Instanz**

- Muss zunächst erzeugt werden
- Kann in jeder Express-App eingehängt werden
- Bei komplexem Routing können Unter-Router erzeugt werden

Router Instanz: Vorteil komplexe URL Struktur

- **Getrennte JavaScript Datei pro Kategorie**
 - z.B. Users.js, Uploads.js, Posts.js
 - Jede Kategorie definiert ihr eigenes URL-Routing
 - Typischerweise im Unterordner „routes“
- **Eine Änderung in „Users.js“ führt nicht zu einer Änderungen in den anderen JavaScript files**
- **Getrennte Verantwortlichkeiten**
 - Weniger Merge-Conflicts in git!

Convention over Configuration Routing

- Typisches Verhalten bei MVC-Frameworks:
 - /\$ControllerName/\$ActionName/\$Id
 - \$ControllerName → Mapt auf Controller Klasse
 - \$ActionName → Mapt auf Methode
 - \$Id → Mapt auf die ID des angefragten Objektes
 - URL-Format typisch bei RESTfull APIs
 - GET /cats/ → Alle Katzen
 - GET /cats/details/12 → Details zu Katze 12
 - GET /cats/create/ → Formular zum Anlegen einer Katze
 - POST /cats/create/ → Legt Katze an

Express Beispiel: Gästebuch

- Im Git Repository zur Vorlesung unter `$Repo/Beispiele/Node.js/GuestbookAPI/`