	Template Engines		AnPr	V 1.0
	Name	Klasse	Datum	

1 SSR vs. CSR

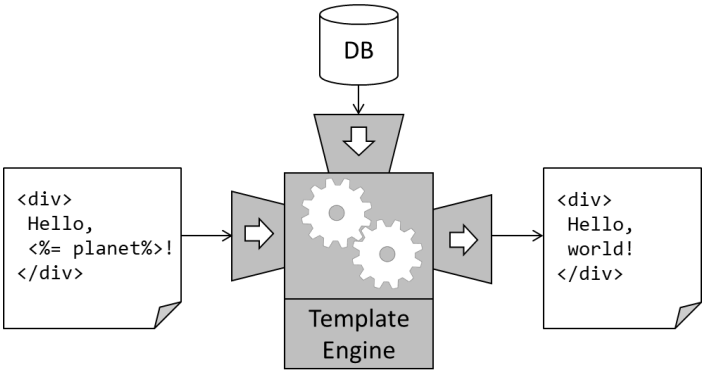
Bevor wir uns um Template Engines kümmern können, müssen wir über die beiden Begriffe SSR und CSR sprechen:

SSR	
CSR	

SSR bedeutet, dass die gesamte HTML-Seite serverseitig vorbereitet wird, bevor sie dem Nutzer per http zugesendet wird. Template Engines sind für das SSR konzipiert und sollen die serverseitige Entwicklungsarbeit verschlanken. Sowohl SSR, als auch CSR haben seine Vorteile. Über CSR werden wir später sprechen. Hier die Vorteile von SSR:

- _____
- _____
- _____

Die wesentliche Grundidee von Template Engines ist es, Strukturinformationen mit Dateninformationen zu vereinen, um daraus eine für den Client lesbare HTML-Seite zu erstellen. Professionelle Engines bieten darüberhinaus noch weitere Funktionalitäten, wie einfache Datenbankanbindungen, Formatierungshilfen etc.



PHP liefert bspw. sehr viele Features, um hierauf schnell eigene Template Engines zu erstellen. Dies erklärt u.A. die große Verbreitung von PHP. Neben eigenen Lösungen gibt es für PHP fertige Engines, wie bspw. „Smarty“ oder „Twig“. Da wir serverseitig mit node.js arbeiten, verwenden wir hier die weit verbreitete Engine EJS.

2 Installation von EJS

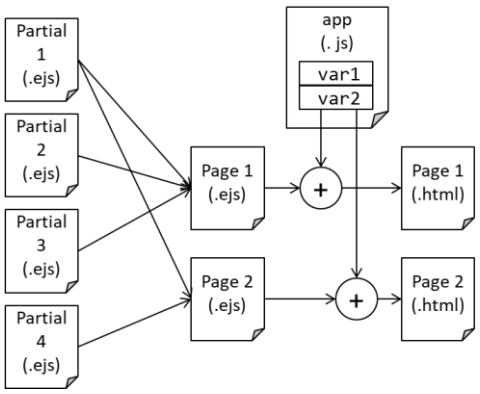
Da EJS im Rahmen von node.js läuft, ist die Installation über den Node Package Manager recht simpel:

```
PS C:\temp\myServer> npm install express ejs
```

Danach sollten die Verzeichnisse für die Seiten und die in den Seiten sich wiederholenden Elementen „partials“ im Rootverzeichnis des Projektes erstellt werden – `views/pages` und `views/partials`. Der Name „`views`“ wird erwartet, „`pages`“ und „`partials`“sind nicht vorgeschrieben, werden aber häufig genutzt.

pages: Hier kommen die `*.ejs` Files rein, welche die Seitenstruktur für das spätere HTML vorgeben
partials: Sich wiederholende Seitenelemente können als „partial“ erstellt und wiederverwendet werden

Darüber hinaus können innerhalb der `*.ejs` Files Variablen in den JavaScript Files (bspw. `app.js`) definiert und in den `*.ejs` ausgewertet werden.



Eine `*.ejs` Datei ist im Wesentlichen ein HTML Format, welches zusätzlich noch Tags für die EJS Verarbeitung eingefügt werden. Diese haben folgendes Schema:

<% Befehlssequenz %>

Für die Tags gibt es verschiedene Optionen. Hier die wichtigsten:

<% … %>	Der Inhalt wird ausgeführt, jedoch nicht angezeigt.
<%= … %>	Der Inhalt wird angezeigt – bspw. der Inhalt einer Variablen
<%- … %>	Der Inhalt wird 1:1 hinzugefügt – ohne HTML Interpretationen. Wird für „includes“ genutzt.

3 Nutzung von EJS in app.js

Wir müssen unserer Applikation mitteilen, dass wir EJS als Rendering Engine nutzen. Hierfür sind zwei Aktionen notwendig:

```
const express = require('express');
const app = express();
const EXP_PORT = 8080;

// Registrieren von EJS als Rendering Engine
app.set('view engine', 'ejs');

// Einbinden der index.ejs Seite für alle GET Anfragen ohne Route:
app.get('/', (req, res) => {
    res.render('pages/index');
});

app.listen(EXP_PORT, () => {
    console.log("Ich höre auf Port! " + EXP_PORT);
});
```

4 Erstellung von Templates

Ein EJS Template ist im Wesentlichen eine HTML-Datei mit der Extension `*.ejs`, welche dynamische Elemente beinhaltet. Hier als Beispiel die `index.ejs` Seite:

```
<!DOCTYPE html>
<html lang="de">
 <head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>EJS Engine</title>
 </head>
 <body>
  Hallo, Welt!
 </body>
</html>
```

Der Inhalt ist 1:1 der gleiche, wie in HTML. Rufen wir nun http://localhost:8080/ auf, sehen wir den Text „Hallo, Welt!“ im Browser.

Nun können wir Informationen von `app.js` in Richtung EJS-File überleiten. Hierzu ändern wir unsere `app.js` wie folgt ab:

```
…
app.get('/', (req, res) => {
    res.render('pages/index', {planet: "Mars"});
});
…
```

Um den Text nun dynamisch einzubinden, müssen wir die `index.ejs` Datei anpassen:

```
…
 <body>
  Hallo, <%= planet %>!
 </body>
</html>
```

Nun sehen wir nach einem Refresh „Hallo, Mars!“.

5 Einbindung von Partials

Wiederkehrende Bereiche, wie bspw. der HTML-Header, können in Partials ausgelagert und referenziert werden. Wir erzeugen also ein `header.ejs` File im `partials` Ordner und verschieben den Header dort hinein:

```
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>EJS Engine</title>
 </head>
```

Somit müssen wir die `index.ejs` Datei um einen Verweis auf `header.ejs` ergänzen:

```
<!DOCTYPE html>
<html lang="de">
<%- include("../partials/header") %>
<body>
  Hallo, <%= planet %>!
 </body>
</html>
```

Variablen können ebenfalls in Partials eingebunden werden. In Pages eingebundene Variablen haben in Partials zwar auch Gültigkeit, wobei es besser ist, sie immer explizit beim include anzugeben. Weiterhin sollte vor allem in den Partials geprüft werden, ob die Variable überhaupt existiert und wenn nicht, sollte ein Defaultwert gesetzt werden. Hierfür bauen wir eine Versionsangabe in den Headertext ein und fügen die Daten über `app.js` ein. Beginnen wir in `app.js`:

```
…
app.get('/', (req, res) => {
    res.render('pages/index', {planet: "Jupiter", version: "1.0"});
});
…
```

Die `version` Variable wird dann von `index.ejs` übernommen und an `hVersion` weitergeleitet:

```
<!DOCTYPE html>
<html lang="de">
<%- include("../partials/header", {hVersion: version}) %>
<body>
```

Im Partial `header.ejs` wird die Variable nun ausgegeben:

```
<head>
 <meta charset="utf-8" />
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
 <title>EJS Engine
   <%= typeof hVersion != 'undefined' ? hVersion : "n.n." %>
 </title>
</head>
```

Über den ternären Operator prüfen setzen wir „n.n“, wenn die Variable `hVersion` nicht existiert. Rufen wir die Seite auf, so können wir oben in der Bezeichnung des Tabs die Version ersehen.

6 Einbindung von JavaScript Code

Ejs Files können JavaScript Code 1:1 verarbeiten. Hierzu nutzen wir die Tags `<%` und `%>`. Sehen wir uns hierzu folgende Anpassung der `index.ejs` Datei an:

```
<!DOCTYPE html>
<html lang="de">
    <%- include("../partials/header", {hVersion: version}) %>
<body>
  Hallo, <%= planet %>!
  <% if(planet == "Welt") { %>
    <p>Be down to earth</p>
  <% } else { %>
    <p>Fly to space</p>
  <% } %>
  </body>
</html>
```

Der JavaScript Code wird also ausgeführt, so dass wir die Ausgaben von HTML Elementen steuern können. Passen wir nun unsere `app.js` an, indem wir zusätzlich ein Array ergänzen:

```
…
app.get('/', (req, res) => {
    res.render('pages/index', {planet: "Mars", version: "1.0",
    trabanten: ["Phobos", "Deimos"]});
});
…
```

In `index.js` können wir nun das Array mit einer `foreach`-Schleife auslesen, welche wir in JavaScript mit einer Callback Methode umsetzen müssen. Üblicherweise erledigt man dies als Lambda-Funktion, damit der Code kompakter wird:

```
…
  <% if (typeof trabanten != 'undefined') { %>
    Trabanten:
    <% trabanten.forEach( trabant => { %>
      <li><%= trabant %></li>
    <% }); %>
  <% } %>
…
```

Auch hier ist es sinnvoll, zuerst die Existenz des Arrays zu überprüfen. Zum Schluss noch ein Hinweis – clientseitige JavaScript Dateien, sowie CSS-, Bild oder ähnliche Dateien müssen über den `public` Ordner als statische Inhalte gesendet werden.