RESTful Web-Services mit ASP.NET Web API

Version 1.3

Überblick

- Architekturkonzepte
- Entwurf von RESTful Web-Services
- Implementierung von Web-Services mit Web API
  - Konfiguration und Routing
  - Implementierung des Controllers
  - (De-)Serialisierung der Ressourcen (Formatter)
  - Content-Negotiation
  - Fehlerbehandlung
- Implementierung von REST-Client
- Swagger

Web-Services: Definition

*"A **Web service** is a software system designed to support **interoperable machine-to-machine interaction** over a **network**. It has an **interface described in a machine-processable format** (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using **SOAP messages**, typically **conveyed using HTTP** with an **XML serialization** in conjunction with other Web-related standards."*

REST ist an HTTP gekoppelt

"We can identify two major classes of Web services:

- ***REST-compliant Web services**, in which the primary purpose of the service is to manipulate representations of Web resources using a uniform set of stateless operations.*
- ***Arbitrary Web services**, in which the service may expose an arbitrary set of operations."*

(Web Services Architecture Document, W3C)

Architekturkonzepte im Web

- Roy Fielding identifizierte in seiner Dissertation (2000) fünf Architekturkonzepte, die das Web erfolgreich machen:
  - *Addressable Resources:* Ressourcen stehen im Mittelpunkt.
  - *A Uniform, Constrained Interface:* Einsatz einer beschränkten Anzahl von Operationen zur Manipulation der Ressourcen.
  - *Representation-Oriented:* Ressourcen stehen in verschiedenen Repräsentationen zur Verfügung.
  - *Communicate Statelessly:* Zustandslose Anwendungen skalieren besser.
  - *Hypermedia As The Engine Of Application State (HATEOAS):* Ressourcen enthalten Links auf mögliche Operationen.
- Diese Konzepte werden unter dem Namen ***Re****presentation* ***S****tate* ***T****ransfer (REST)* zusammengefasst.

Architekturkonzepte von Web-Services

- Methoden-zentriert → SOAP-basierte Web-Services
  - Basiskonzept: Interfaces mit vielen Methoden
  - Daten (Ressourcen) werden als in Form von Parametern an Methoden übergeben bzw. von diesen zurückgegeben.
  - Technische Umsetzung:
    - Zerlegung der Methode in Anfrage- und Antwort-Dokument
    - Serialisierung der Eingangs- und Rückgabeparameter
- Ressourcen-zentriert → REST
  - Basiskonzept:
    - Ressourcen stehen im Mittelpunkt und werden zwischen Client und Service ausgetauscht.
    - Auf Ressourcen kann eine vorgegebenen Menge an Operationen angewandt werden.
  - Technische Umsetzung:
    - Austausch der Ressourcen über HTTP-Anfragen und -Antworten
    - Abbildung der Operationen auf HTTP-Methoden

Uniform, Constrained Interface

- Für jede Ressource steht eine beschränkte Anzahl von Operationen zur Verfügung (HTTP-Methoden):
  - **GET:** Lesen der Ressource. Idempotent und sicher (Zustand wird nicht verändert).
  - **PUT:** Aktualisieren der übergebenen Ressource (speichern bzw. einfügen, falls nicht vorhanden). Idempotent: Mehrmaliges Speichern ändert den Zustand der Ressource nicht.
  - **POST:** Hinzufügen der übergebenen Ressource. *Nicht idempotent:* Jede POST-Operation verändert Zustand.
  - **DELETE:** Löschen der Ressource. Idempotent.
  - **HEAD:** Wie GET, jedoch werden nur Rückgabe-Codes geliefert.
  - **OPTIONS:** Welche Methoden unterstützt eine Ressource.

Uniform, Constrained Interface – Vorteile

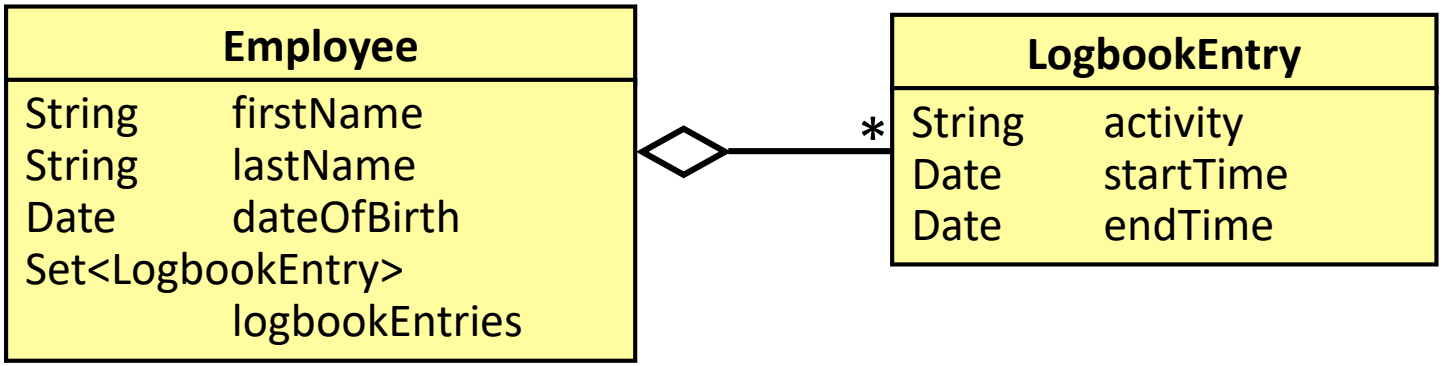
- Einfachheit
  - Interface ist leichter zu verstehen.
  - Bedarf für Service-Metadaten (Interface-Beschreibung) ist geringer.
  - Keine Proxys und keine komplexe Client-Bibliothek notwendig.
- Interoperabilität
  - HTTP-Client-Bibliothek ist ausreichend und existiert praktisch für jede Programmiersprache.
  - Keine Probleme mit Hersteller-Interoperabilität (Kompatibilitäts-probleme bei Implementierungen von WS-*-Protokollen)
- Skalierbarkeit
  - Ergebnisse von lesenden Operationen können zwischengespeichert (Caching) werden: Client-seitig oder von Proxy-Servern.

Representation-Oriented

- Client und Server tauschen mit HTTP-Anfragen und -Antworten Repräsentationen der Ressourcen untereinander aus.
- Die Repräsentation der Daten (Ressourcen) ist im Gegensatz zu SOAP-basierten Web-Services nicht standardisiert.
- Verbreitete Datenformate:
  - XML (Java)
  - JSON (JavaScript)
  - YAML (Perl, Python, Ruby)
- Mit dem HTTP-Header *Content-Type* wird definiert, in welcher Repräsentation die Ressource übertragen wird.
- Im HTTP-Header *Accept* kann der Client das gewünschte Datenformat angeben (z.B. *Accept: application/xml*).

Entwurf von RESTful Web-Services (1)

- Festlegen des Objektmodells (Domänenklassen)



- Modellierung der URIs
  - `/employees`: alle Angestellten
  - `/employees/{id}`: ein einzelner Angestellter
  - `/logbookentries`: alle Arbeitszeiteinträge
  - `/logbookentries/{id}`: ein Arbeitszeiteintrag
  - `/employees/{id}/logbookentries`: alle Arbeitszeiteinträge eines Angestellten.

Entwurf von RESTful Web-Services (2)

- Definition der Datenformate

```
<employee id="1">
  <firstname>Roy</firstname>
  <lastname>Fielding</lastname>
  <dob>1965-01-10</dob>
  <logbookentries>
    <logbookentry id="123">
      <link rel="self" href="http://.../logbookentries/123" />
      <activity>Testing</activity>
      <startTime>2010-10-01 08:00</startTime>
      <endime>2010-10-01 12:15</endTime>
    </logbookentry>
  </logbookentries>
</employee>
```

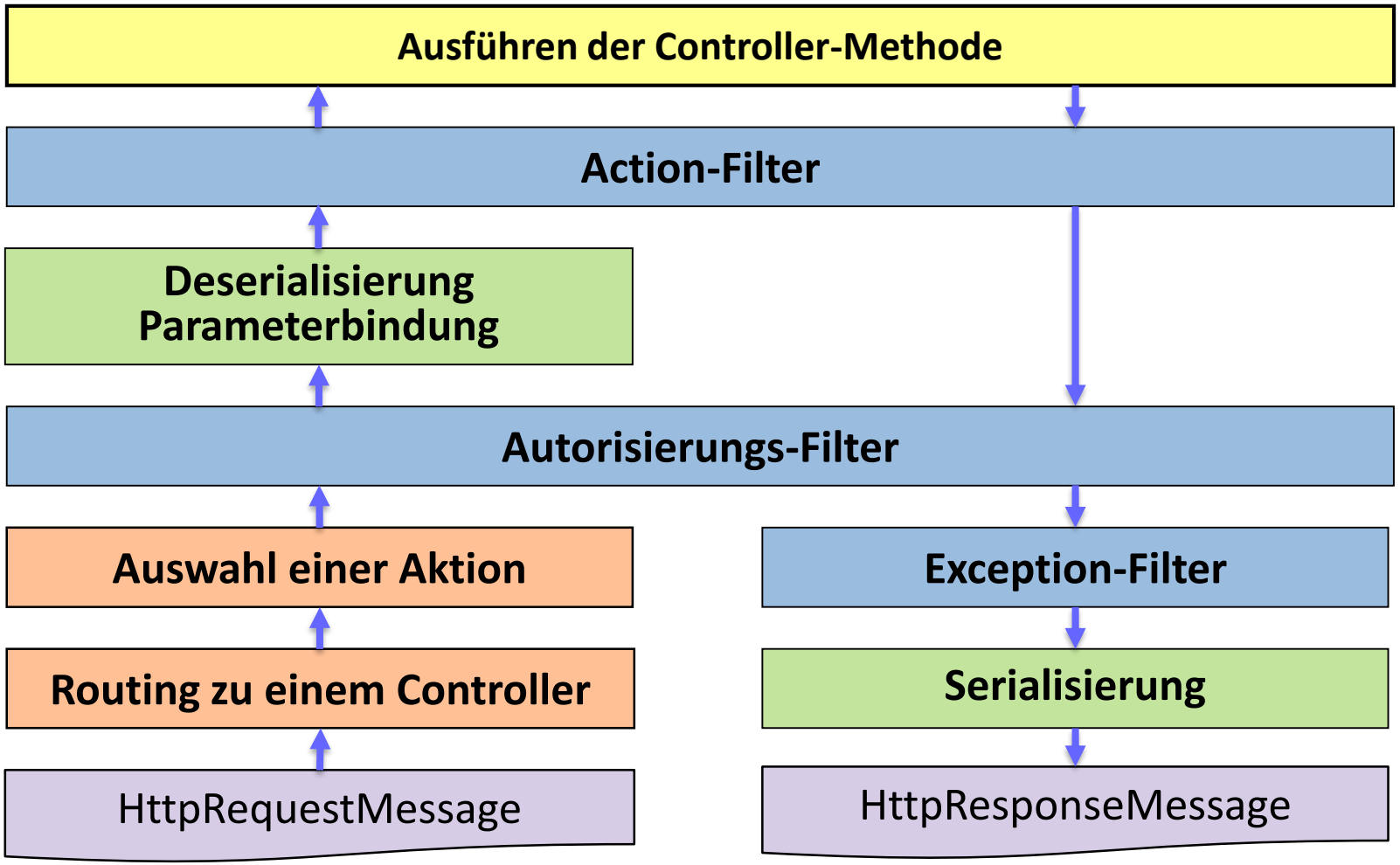
  - Bei POST-Anfragen entfällt das Attribut *id*.
  - Links erlauben es dem Client, mit der entsprechenden Ressource zu interagieren.
  - Mit Links kann auch die Länge der übertragenen Nachricht verringert werden (z. B. Realisierung von Paging)

Entwurf von RESTful Web-Services (3)

- Definition der Semantik der HTTP-Methoden
  - GET `/employees[?startIndex=0&size=5]`:
    - Lesen (eines Teils) aller Angestelltenobjekte.
  - GET `/employees/{id}`:
    - Lesen eines Angestelltenobjekts über dessen Schlüssel.
  - POST `/employees`:
    - Einfügen eines Angestelltenobjekts. Schlüssel wird generiert und zurückgegeben.
  - PUT `/employees/{id}`:
    - Aktualisierung eines Angestelltenobjekts. Objekt neu anlagen, falls nicht vorhanden.
  - DELETE `/employees/{id}`:
    - Löschen eines Angestelltenobjekts (oder als gelöscht markieren).

Implementierung von REST-Services mit ASP.NET Web API

Das Laufzeitsystem von ASP.NET Web API



Konfiguration eine Web-API-Anwendung

- Globale Parameter werden in einem globalen Konfigurationsobjekt vom Typ *HttpConfiguration* gespeichert.
- Das Konfigurationsobjekt wird beim Starten der Anwendung initialisiert.

```
public class MyWebApiApplication : System.Web.HttpApplication {
  protected void Application_Start() {
    HttpConfiguration config = GlobalConfiguration.Configuration;
    config.Routes.MapHttpRoute(…);
    config.Filters.Add(…);
    config.Formatters.Add(…);
    config.Services.Add(…);
    config.DependencyResolver = …;
    config.MessageHandlers.Add(…);
  }
}
```

Konfiguration des Routing-Systems

```
protected void Application_Start() { // wird beim Starten der Web-Anw. aufgerufen
  GlobalConfiguration.Configure(WebApiConfig.Register);
}
public static void Register(HttpConfiguration config) {
  config.Routes.MapHttpRoute(
    name: "WorklogRoute",                          // Name der Route
    routeTemplate: "worklog/{controller}/{id}",    // URL mit Parametern
    defaults: new { controller="Home", id = RouteParameter.Optional }
                                                   // Standardwerte für Parameter
  );
}
```

URL	Abgebildet auf
/worklog	{ controller="Home", id=null }
/worklog/employees	{ controller="Employees", id=null }
/worklog/employees/1023	{ controller="Employees", id=1023 }

Controller

- Jede HTTP-Anfrage wird an einen Controller weitergeleitet.
- An die Controller werden die Anfragedaten in aufbereiteter Form übergeben.
  - URL-Parameter
  - Header
  - Body der Anfrage
- Controller stellen die Verbindung zur Geschäftslogik her.
- Controller bereiten die Daten für die HTTP-Antwort auf.
  - HTTP-Statuscodes
  - Header
  - Body der Antwort
- Controller-Methoden eigenen sich gut für Unit-Tests.

Auswahl von Controller-Methoden

```
public class EmployeesController : System.Web.Http.ApiController {
  public IEnumerable<Employee> GetAll() { … }
  public Employee              Get(int id) { … }
  [HttpPut]
  public void Update(int id, Employee empl) { … }
  public void Delete(int id) { … }
}
```

- Die HTTP-Methode der Anfrage legt fest, welche Controller-Methode aufgerufen wird:
  - Präfix im Methodennamen gibt an, welcher HTTP-Methode die Controller-Methode zugeordnet ist.
  - Methode mit passenden Parametertypen wird ausgewählt.
- Über Attribute kann Zuordnung explizit definiert werden: *[HttpGet], [HttpPut], [HttpPost], [HttpDelete]* etc.

Attribut-basiertes Routing

- Die Zuordnung einer Controller-Methode zu einer URL kann auch über Attribute erfolgen:

```
[RoutePrefix("worklog")]
public class EmployeesController : System.Web.Http.ApiController {
  [Route("employees")]
  public IEnumerable<Employee> FindAllEmployees() { … }
  [Route("employees/{id:int}")]
  public IEnumerable<Employee> FindEmployee(int id) { … }
}
```

- In diesem Fall kann die Definition von Routen entfallen.
- Attributbasiertes Routing muss aber explizit aktiviert werden:

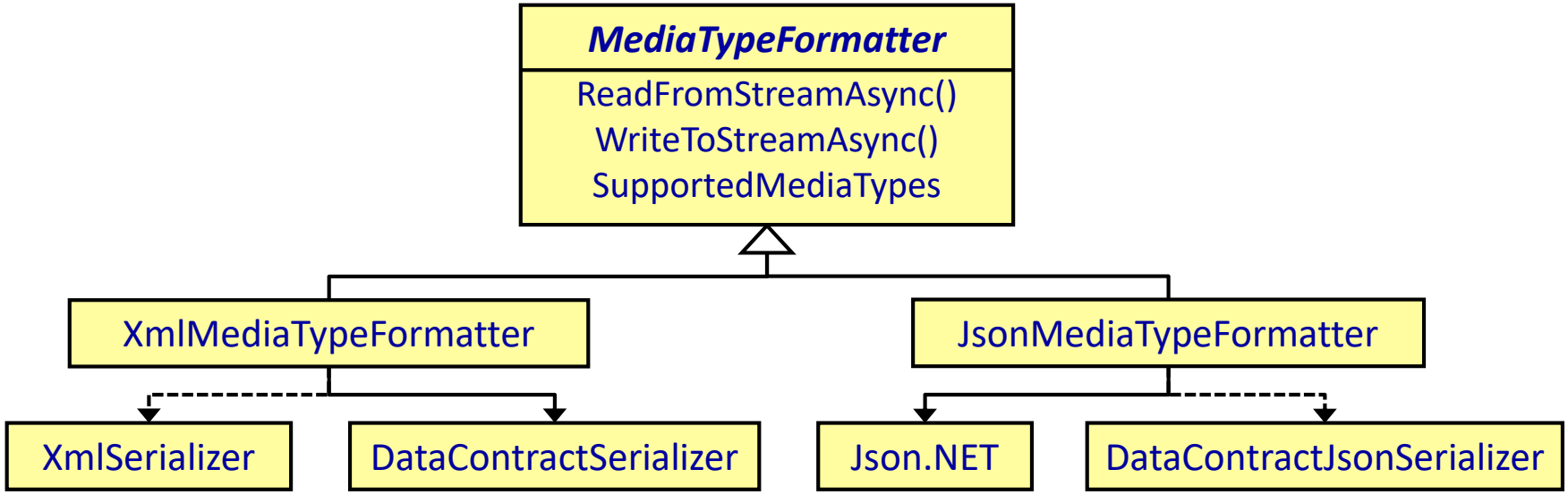
```
public static void Register(HttpConfiguration config) {
  config.MapHttpAttributeRoutes();
}
```

Parameterbindung

```
public class EmployeesController : System.Web.Http.ApiController {
  [HttpGet]
  public IEnumerable<Employee> GetAll([FromUri] int skip,
                                      [FromUri] int count = 10) { … }
  [HttpPost]
  public HttpResponseMessage Add([FromBody] Employee empl) { … }
}
```

- Einfache Datentypen (int, double, string, DateTime, …) werden standardmäßig aus der URI übernommen.
- Komplexe Datentypen werden standardmäßig aus dem Körper der Anfrage übernommen (und deserialisiert).
- Mit *[FromUri]* und *[FromBody]* kann man Standard überschreiben.
- Mit benutzerdefinierten Value-Providern (Interface *IValueProvider*) kann man auch andere Quellen für Parameter nutzen (z. B. Header).

Formatter: (De-)Serialisierung der Ressourcen



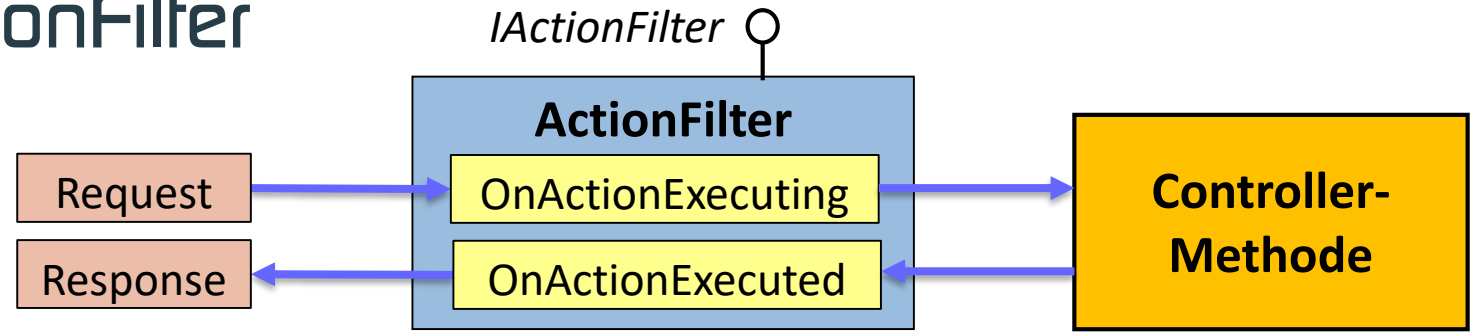
- Die (De-)Serialisierung erfolgt mit Klassen vom Typ *MediaTypeFormatter.*
- Die Web-API stellt Formatter für XML und JSON zur Verfügung.
  - DataContract/DataMember-Attribute werden erkannt (auch von Json.NET)
- Formatter für andere Repräsentationsarten können hinzugefügt werden.
  - Neue Formatter-Klasse von *MediaTypeFormatter* ableiten.
  - Formatter in globaler Konfiguration registrieren.

Parametervalidierung

```
[HttpPost]
public HttpResponseMessage AddEmployee([FromBody] Employee empl) {
  if (ModelState.IsValid) {
    …
  }
  else
    return Request.CreateErrorResponse(HttpStatusCode.BadRequest,
                                       ModelState);
}
```

- In *ModelState* wird Fehlerinformation gespeichert, die bei der Parameterbindung und beim Deserialisieren der Payload entsteht.
- *ModelState* enthält Fehlerinformationen zu den Methodenparametern, z. B. *ModelState["empl"]*
- Bei jeder Anfrage sollte überprüft werden, ob ein Validierungsfehler vorliegt → Statuscode 400 (Bad Request) zurückgeben.

ActionFilter



Beispiel: ValidationFilter

```
public class ValidationActionFilter : ActionFilterAttribute {
  public override void OnActionExecuting(HttpActionContext actionCtx) {
    var modelState = actionCtx.ModelState;
    if (! modelState.IsValid)
      actionCtx.Response = actionCtx.Request
        .CreateErrorResponse(HttpStatusCode.BadRequest, modelState);
  }
  public void OnActionExecuted(HttpActionExecutedContext ctx) { … }
}
```

```
[ValidationActionFilter]
public class EmployeesController : ApiController { … }
```

Ergebnisse von Controller-Methoden

```
[HttpGet]
public Employee Get(int id) { … }

[HttpPost]
public HttpResponseMessage AddEmployee([FromBody] Employee empl) {
  var response = Request.CreateResponse<Employee>(
                  HttpStatusCode.Created, empl);
  string uri = Url.Link("WorklogRoute", new { id = empl.Id });
  response.Headers.Location = new Uri(uri);
  return response;
}
```

- Der Body der HTTP-Antwort wird durch den Rückgabewert der Controller-Methode definiert.
- Man kann auch *HttpResponseMessage* retournieren:
  - *Content:* Payload der HTTP-Antwort kann definiert werden.
  - *StatusCode:* HTTP-Statuscode kann explizit gesetzt werden.
  - *Headers:* Header-Element können gesetzt/hinzugefügt werden.

Content Negotiation

- Ein RESTful Service kann verschiedene Repräsentationen einer Ressource unterstützen (XML, JSON, YAML, …).
- Die Repräsentation einer Ressource wird im HTTP-Header *Content-Type* als *Mime-Typ* angegeben:

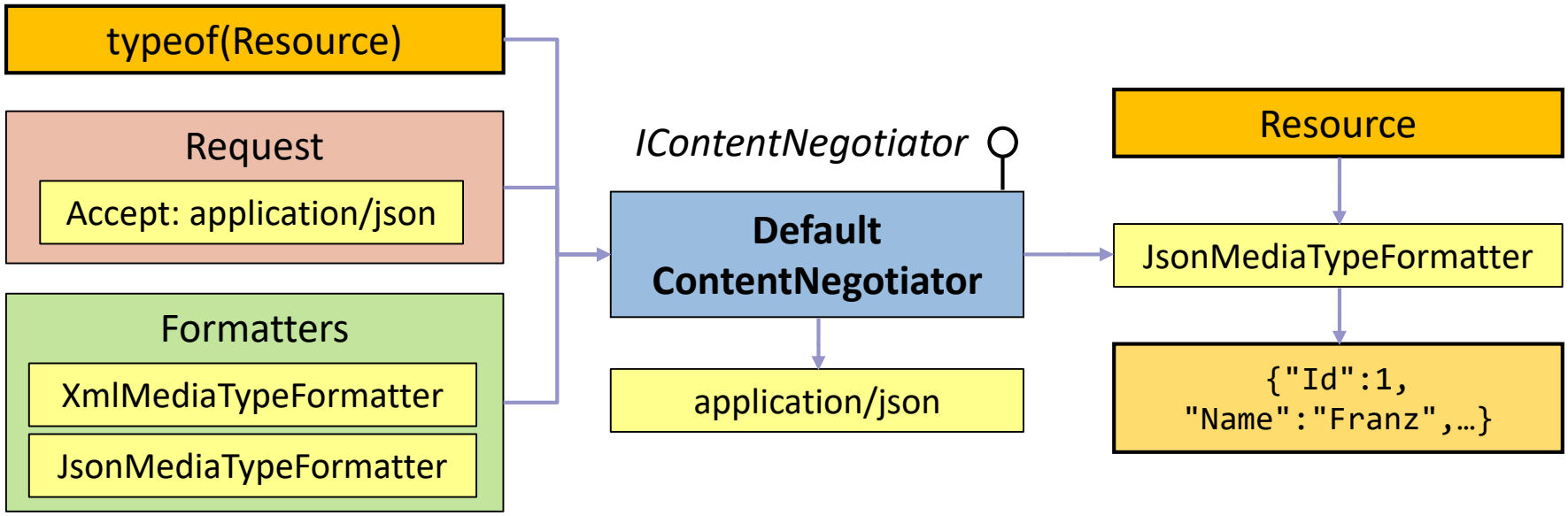
```
Content-Type: application/xml
```

- Der Client kann im HTTP-Header *Accept* definieren, welche Repräsentationen er akzeptiert bzw. bevorzugt:

```
Accept: application/*, text/*;q=0.9, application/json
```

  - Je spezifischer ein Mime-Typ ist, desto höher ist seine Priorität.
  - Mit Qualifizierern kann die Priorität verringert werden (Default: q=1.0)
- Auch die Art der Komprimierung *(Accept-Encoding)* und die Sprache *(Accept-Language)* kann zwischen Client und Service verhandelt werden.

Serialisierung der HTTP-Antwort/Content-Negotiation



- Die Laufzeitumgebung ermittelt mithilfe eines *ContentNegotiators*, mit welchem *MediaTypeFormatter* die Serialisierung der Ressource durchgeführt werden soll.
  - Für XML und JSON existieren bereits Implementierungen von *MediaTypeFormatter*.
  - *MediaTypeFormatter* werden auch zum Deserialisieren von Eingangsparametern verwendet.
  - Man kann benutzerdefinierte *MediaTypeFormatter* hinzufügen.
  - Auch der *ContentNegotiator* kann ausgetauscht werden.

Fehlerbehandlung

- Wird eine Ausnahme nicht behandelt, wird Statuscode 500 (Internal Server Error) retourniert.
- Mit *HttpResponseException* kann der HTTP-Statuscode definiert werden.
- Mit *HttpResponseMessage* kann zusätzlich der Fehler näher beschrieben werden.

```
[HttpGet]
public Employee GetEmployee(int id) {
  Employee empl = DaoFactory.EmployeeDao.FindById(id);
  if (empl == null) {
     var msg =
       new HttpResponseMessage(HttpStatusCode.NotFound) {
         Content = new StringContent("Employee " + id + "not found")
       }
     throw new HttpResponseException(msg);
  }
  return empl;
}
```

Fehlerbehandlung: Exception-Filter

- Mit Exception-Filtern kann definiert werden, wie nicht behandelte Ausnahmen auf die HTTP-Antwort abgebildet werden:

```
public class NotImplExceptionFilterAttribute :
                ExceptionFilterAttribute {
  public override void OnException(HttpActionExecutedContext ctx) {
    if (ctx.Exception is NotImplementedException)
      ctx.Response =
        new HttpResponseMessage(HttpStatusCode.NotImplemented);
  }
}
```

- Exception-Filter können Controllern oder Controller-Methoden zugeordnet werden:

```
[NotImplExceptionFilter]
public class EmployeesController : ApiController { … }
```

- Oder global allen Controllern:

```
GlobalConfiguration.Configuration.Filters.Add(
    new NotImplExceptionFilterAttribute());
```

Häufig verwendetet Statuscodes

- *200 (OK):* Anfrage wurde erfolgreich verarbeitet (bei GET).
- *201 (Created):* Ressource wurde erfolgreich erzeugt.
- *204 (No Content):* Anfrage wurde erfolgreich verarbeitet und Körper der Antwort ist leer (bei PUT, DELETE).
- *400 (Bad Request):* Anfrage ist fehlerhaft aufgebaut.
- *403 (Forbidden):* Client hat nicht die Rechte, auf die Ressource zuzugreifen.
- *404 (Not Found):* URI ist keiner Servicemethode zugeordnet.
- *405 (Method Not Allowed):* URI ist einer Servicemethode zugeordnet, die aber HTTP-Methode nicht unterstützt.
- *406 (Not Acceptable):* Angeforderter Medientyp wird nicht unterstützt.
- *409 (Conflict):* Inkonsistenter Zustand, Ressource wird beispielsweise ein zweites Mal hinzugefügt.
- *415 (Unsupported Media Type):* Übergebener Medientyp nicht unterstützt.

Authentifizierung und Autorisierung

- Die Authentifizierung kann im Host erfolgen.
  - Der IIS stellt dafür entsprechende Module zur Verfügung.
- Die Authentifizierung kann aber auch in einem *HttpMessageHandler* durchgeführt werden.
- Autorisierungs-Filter
  - Führt Methode nur dann aus, wenn entsprechende Rechte vorliegen.
  - Autorisierungsfilter muss aktiviert werden (global, auf Controller- oder Methodenebene).

```
[AllowAnonymous]
public Employee Get(int id) { … }

[HttpPost] [Authorize(Roles="Administrators")]
public HttpResponseMessage AddEmployee(Employee empl) { … }
```

  - Benutzerdefinierte Autorisierungs-Filter sind möglich.

Hosting

- Die Web-API unterstützt zwei Hosting-Varianten:
  - Web-Hosting: Nutzt Hosting-Infrastruktur von ASP.NET.
  - OWIN Self-Hosting: Service wird in Windows-Prozess gehostet.
- Self-Hosting

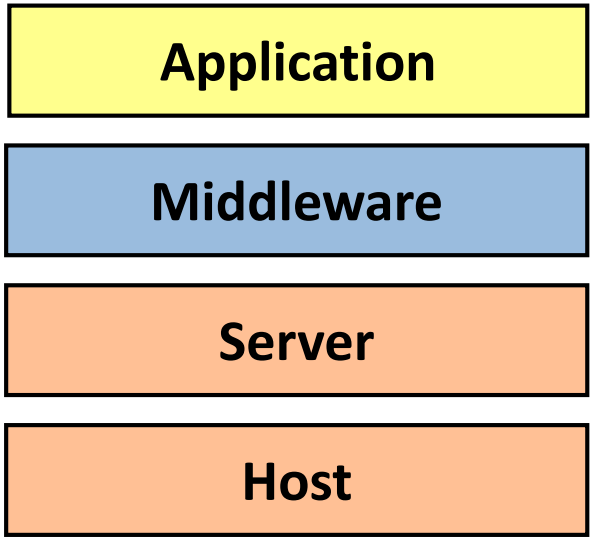
```
public class Startup {
  public void Configuration(IAppBuilder appBuilder) {
    HttpConfiguration config = new HttpConfiguration();
    config.Routes.MapHttpRoute(name: "WorklogRoute",
                               routeTemplate: "worklog/{controller}");
    appBuilder.UseWebApi(config);
  }
}
static void Main(string[] args) {
  using (WebApp.Start<Startup>(url: "http://localhost:9000")) { … }
}
```

Exkurs: OWIN und Katana (1)

- ASP.NET ist über die Jahre gewachsen und die Basis für viele Komponenten:
  - ASP.NET Web-Forms
  - ASP.NET MVC
  - ASP.NET Web API, SignalR, …
- Monolithischer Aufbau und fixer Bestandteil des .NET-Frameworks → lange Entwicklungszyklen.
- OWIN: Open Web Interface for .NET
  - Community-getriebener Standard.
  - Einfache Erweiterbarkeit um zusätzliche Komponenten.
  - Einfache Portierbarkeit von Web-Anwendungen (unterschiedliche Hosts, Plattformen etc.).
- Katana: Verschiedene Implementierungen von OWIN durch Microsoft.

Exkurs: OWIN und Katana (2)

- 4-schichtige Architektur.
- **Host:** Orchestrierung der Request-Pipeline.
  - IIS
  - Custom (Self-Hosting),
  - OwinHost.exe (Konsole).
- **Server:** Horchen auf Requests und Weiterleiten an Pipeline.
  - Microsoft.Owin.Host.SystemWeb
  - Microsoft.Owin.Host.HttpListener
- **Middleware:** Komponente der Pipeline, die Requests verarbeitet
  - Erhält Request/Response-Kontext
  - Gibt Kontrolle an nachgelagerte Middleware-Komponente weiter.
- **Application:** MVC-, Web-API, SignalR-Anwendung.



Exkurs: OWIN und Katana (3)

- Beispiel für Orchestrierung verschiedener Middleware-Komponenten:

```
public class Startup {
  public void Configuration(IAppBuilder appBuilder) {
    appBuilder.UseOAuthAuthorizationServer(
      new OAuthAuthorizationServerOptions { … })
    appBuilder.UseCors(CorsOptions.AllowAll);
    appBuilder.MapSignalR(new HubConfiguration { … });
    appBuilder.UseWebApi(new HttpConfiguration { … });
  }
}
static void Main(string[] args) {
  using (WebApp.Start<Startup>(url: BASE_ADDRESS) { … }
}
```

Implementierung von REST-Clients mit .NET

RESTful .NET-Clients: HTTP-Bibliothek

- Das .NET-Framework bietet eine HTTP-Bibliothek, die zur Implementierung von Clients für REST-Services verwendet werden kann.
- Nachteil: Implementierung ist relativ aufwändig:
  - (De-)Serialisierung
  - Fehlerbehandlung: Fehlercodes führen zu Ausnahmen

```
public Employee GetEmployee(int id) {
  HttpWebRequest request = (HttpWebRequest)WebRequest.Create(
                                new Uri(baseUri, "worklog/employees/" + id));
  request.Method = "GET";
  request.ContentType = "application/xml";

  Employee employee;
  using (HttpWebResponse response =(HttpWebResponse)request.GetResponse())
  using (Stream responseStream = response.GetResponseStream()) {
    DataContractSerializer ser = new DataContractSerializer(typeof(Employee));
    employee = (Employee)ser.ReadObject(responseStream);
  }
  return employee;
}
```

RESTful .NET-Clients mit Web API

- Die Web-API bietet eine ausgezeichnete Unterstützung für die Implementierung von Clients für REST-Services:

```
public Employee GetEmployee(int id) {
  HttpClient httpClient = new HttpClient();
  httpClient.DefaultRequestHeaders.Accept.Add(
    new MediaTypeWithQualityHeaderValue("application/xml"));
  HttpResponseMessage resp = await httpClient.GetAsync(
        new Uri(baseUri, "worklog/employees/" + id);
  resp.EnsureSuccessStatusCode();
  return await resp.Content.ReadAsAsync<Employee>();
}
```

- Vorteile:
  - Automatische (De-)Serialisierung mithilfe von Formattern *(MediaTypeFormatter).*
  - Asynchrone Methoden.
  - Zugriff auf HTTP-Header und HTTP-Statuscodes.
  - Komfortmethoden für Fehlerbehandlung.

RESTful JavaScript-Client

- JavaScript-Clients profitieren vor allem von der JSON-Repräsentation der Ressourcen (JSON = JavaScript Object Notation).
- Mithilfe der JavaScript-Funktion *eval* kann eine Zeichenkette in ein JavaScript-Objekt konvertiert werden.
- Problematisch sind die vielen JSON-Formate (mapped, natural, badgerfish, …)

```
function updateEmployeeInfo(custId) {
  var url = 'http://…/worklog/employees/' + custId;
  http_request = getXMLHttpRequest();
  http_request.open('GET', url, true);
  http_request.setRequestHeader("Content-Type", 'application/json');
  http_request.onreadystatechange = displayEmployeeInfo;
  http_request.send(null);
}
function displayEmployeeInfo() {
  if (http_request.readyState == 4 && http_request.status == 200) {
    var employee = JSON.parse(http_request.responseText);
    var lastName = employee.lastname;
    …
  }
}
```

RESTful jQuery-Client

- jQuery stellt Methoden zum asynchronen Aufruf von RESTful Web-Services zur Verfügung:
  - `$.ajax({ url: "…", accepts: "…", dataType: "…", type: "GET" | …, success: function(…) { … }, error: function(…) { … } });`
  - `$.get(url, function(…) { … }, type).fail(function(…) { … })`
  - `$.post(url, data, function(…) { … }, type).fail(function(…) { … })`

```
$(document).ready(function(){
  $("#employeeID").keyup(function() {
    var id = $("#employeeID").val();
    $.get("http://localhost:8000/worklog/employees/" + id,
      function(employee) {
        $("#tdFirstName").text(employee.firstname);
        $("#tdLastName").text(employee.lastname);
        $("#tdDob").text(formatDate(new Date(employee.dob)));
      }, "json").fail(function() {
        …
      });
  });
});
```

Angular-Client (1)

- Der Zugriff auf den REST-Service wird am besten in ein Angular-Service ausgelagert.
  - Die Schnittstelle ist typsicher
  - Der Zugriff erfolgt asynchron (Observable).

```
export class Employee {
  Id: number;
  FirstName: string;
  ...
}
```

```
import { HttpClient } from "@angular/common/http";
import { Observable } from 'rxjs';

@Injectable()
export class WorkLogService {
  constructor(private httpClient: HttpClient) { }
  getById(id: number): Observable<Employee> {
    return this.httpClient.get<Employee>(`${environment.workLogUrl}/${id}`);
  }
}
```

Angular-Client (2)

- Verwendung des Angular-Service:

```
import { WorkLogService } from './worklog.service'
import { Employee } from './worklog.service'

@Component({…})
export class AppComponent implements OnInit {

  currentEmployee: Employee = null;

  constructor(private workLogService: WorkLogService) { }

  updateEmployee(id: number): void {
    this.workLogService.getById(id)
        .subscribe(empl => this.currentEmployee = empl);
  }
}
```

Swagger

Swagger

- Swagger erlaubt eine sprachunabhängig Beschreibung der Schnittstelle zu REST-Services.
- Swagger-Dokumente werden im JSON-Format repräsentiert und sind menschen- und maschinenlesbar.
- *OpenAPI Specification:* Formale Spezifikation für Swagger-Dokumente.
- Werkzeuge:
  - **Swagger Editor:** Input: API-Beschreibung als YAML-Dokument; Output: Swagger-Dokument
  - **Swagger UI:** HTML-, CSS- und JavaScript-Artefakte zum Generieren einer optisch gut aufbereiteten Dokumentation von REST-APIs
  - **Swagger CodeGen:** Erzeugung von Client-Proxys und Server-Stubs aus Swagger-Dokument: Java (JAX-RS), PHP,Node.js, Haskell, Spring MVC, ASP.NET 5.
  - **NSwag/AutoRest:** Code-Generierung für .NET
  - **Swagger-Core:** Java-Bibliotheken zum Lesen und Generieren von Swagger-Dokumenten
  - **NSwag/Swashbuckle:** .NET-Bibliotheken zum Lesen und Generieren von Swagger-Dokumenten

Swagger-Schema – Beispiel (1)

```
{
  "swagger": "2.0",
  "info": {
    "version": "v1",
    "title": "Worklog API"
  },
  "basePath": "/",
  "paths": {
    "/worklog/Employees": {
      "get": { ... },
      "post": { ... }
    },
    "/worklog/Employees/{id}": {
      "get": { ... },
      "put": { ... },
      "delete": { ... }
    }
  },
  "definitions": { ... },
}
```

```
"/worklog/Employees/{id}": {
  "get": {
    "operationId": "GetById",
    "produces": ["application/json", …],
    "parameters": [{
      "name": "id",
      "in": "path",
      "required": true,
      "type": "integer",
      "format": "int32"
    }],
    "responses": {
      "200": {
        "description": "OK",
        "schema": {
          "$ref": "#/definitions/Employee"
        }
      },
      "404": { "description": "NotFound" }
    }
  }
}
```

Swagger-Schema – Beispiel (2)

```
"/worklog/Employees/{id}": {
  "get": {
    "operationId": "GetById",
    "produces":["application/json", …],
    "parameters": [ … ],
    "responses": {
      "200": {
        "description": "OK",
        "schema": {
          "$ref": "#/definitions/Employee"
        }
      },
      "404": { "description": "NotFound" }
    }
  }
}
```

```
"definitions": {
  "Employee": {
    "type": "object",
    "properties": {
      "Id": {
        "format": "int32",
        "type": "integer"
      },
      "FirstName": {
        "type": "string"
      },
      "LastName": {
        "type": "string"
      },
      "DateOfBirth": {
        "format": "date-time",
        "type": "string"
      }
    }
  }
}
```

NSwag – Integration in Web API

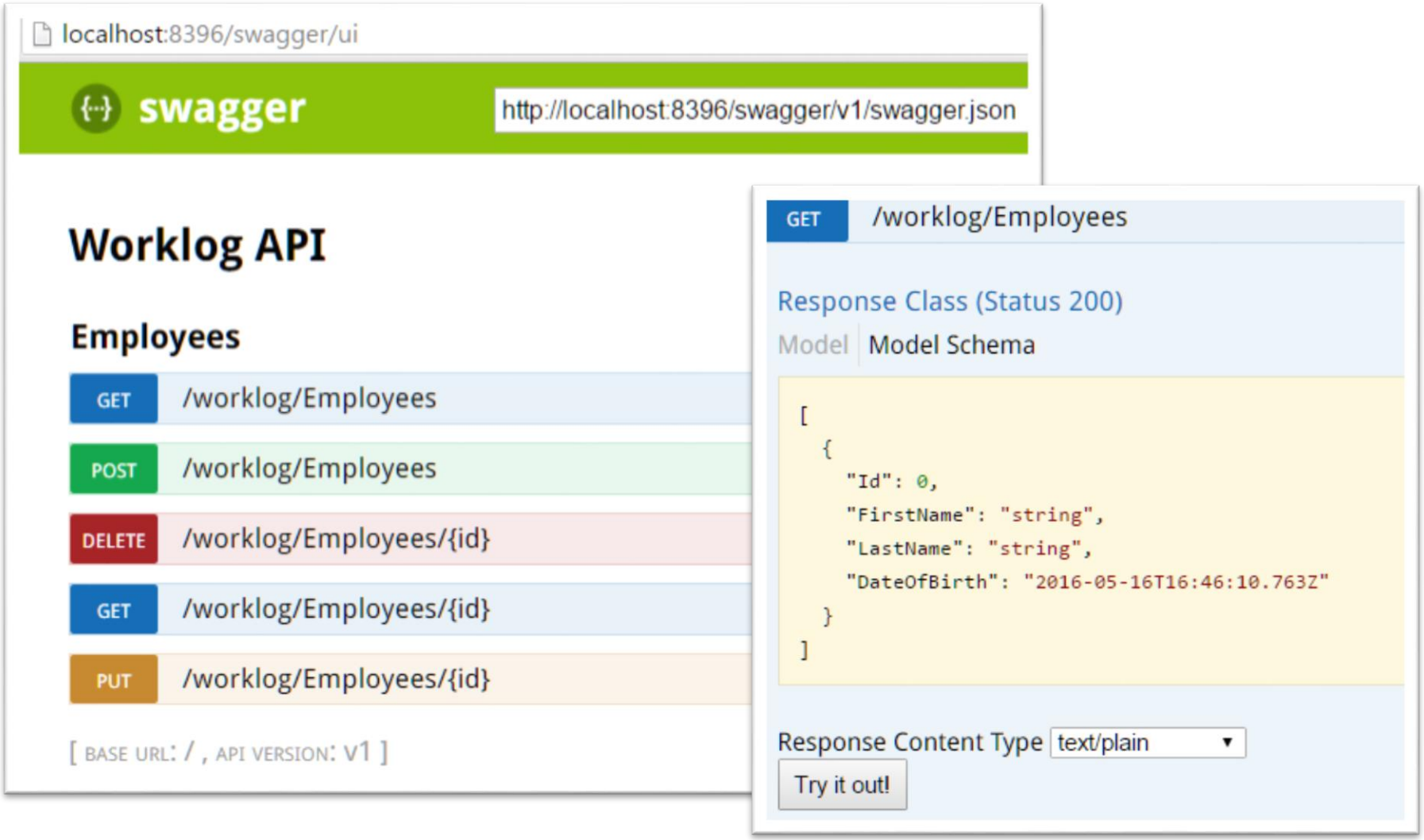
- Generierung von Swagger-Metadaten und Swaggger-UI aktivieren:

```
public class SwaggerConfig {
  public static void RegisterSwagger(RouteCollection routes) {
    routes.MapOwinPath("swagger", app => {
      app.UseSwaggerUi3(typeof(WebApiApplication).Assembly, settings => {
        settings.GeneratorSettings.Title = "WorkLog API";
        settings.MiddlewareBasePath = "/swagger";
      });
    });
  }
}
```

- Service-Implementierung mit Metadaten anreichern:

```
[HttpGet]
[Route("worklog/{id}")]
[SwaggerOperation(operationId: nameof(GetById))]
[SwaggerResponse(typeof(Employee))]
[SwaggerResponse(HttpStatusCode.NotFound, typeof(void))]
public Employee GetById(int id) { … }
```

Swagger UI


localhost:8396/swagger/ui
swagger
http://localhost:8396/swagger/v1/swagger.json
Worklog API
Employees
GET /worklog/Employees
POST /worklog/Employees
DELETE /worklog/Employees/{id}
GET /worklog/Employees/{id}
PUT /worklog/Employees/{id}
[BASE URL: / , API VERSION: V1]
GET /worklog/Employees
Response Class (Status 200)
Model Model Schema
[
{
"Id": 0,
"FirstName": "string",
"LastName": "string",
"DateOfBirth": "2016-05-16T16:46:10.763Z"
}
]
Response Content Type text/plain
Try it out!