Ein Set-up für REST-Schnittstellen

Die Eleganz von REST APIs mit Symfony

Wir EntwicklerInnen wollen klar definierte Schnittstellen und finden Dokumentieren mühsam. Wir lieben das Gefühl valider Daten, haben aber keine Lust, langweiligen Code für die Validierung zu schreiben. Ja zu sauberen Schnittstellen, Nein zu Duplikaten im Code. Das Ziel ist es, möglichst einfach sicherzustellen, dass alle Schnittstellen exakt definiert sind und nur Daten in der vorgegebenen Form bei der Applikation ankommen.

von Sabine Bär

Was ist nötig, um klar definierte Schnittstellen zu erhalten, ohne aufwendig dokumentieren oder langweiligen Code schreiben zu müssen? Wie entsteht sauberer und zweckmäßiger Code ohne Duplikate? Diese Gedanken sind mir durch den Kopf gegangen, als ich wieder einmal ein Set-up für REST APIs mit dem Framework Symfony aufgebaut habe. Dieses Mal wollte ich alle diese Punkte unterbringen. Zur Spezifikation von APIs verwenden wir schon lange das OpenAPI-Format (ehemals Swagger). Damit sind die Schnittstellen dokumentiert und können einfach getestet werden. Die Informationen, die für die Validierung von eingehenden Daten benötigt werden, sind alle schon in dieser Spezifikation vorhanden und warten nur darauf, wiederverwendet zu werden. Auch die nachfolgende Weiterverarbeitung der Daten in eine nutzbare Form soll möglichst einfach funktionieren. Mit verschiedenen Symfony Bundles kann ein Set-up geschaffen werden, das genau das macht und einem bei jedem neuen API, das man implementiert, ein richtig gutes Gefühl gibt.

OpenAPI vs. Swagger

2015 wurde aus der Swagger-Spezifikation der offene Standard OpenAPI [1], an dem seither ein Konsortium aus verschiedenen Experten gemeinsam arbeitet. Das war ein wichtiger Schritt, hat allerdings zu einigen Verwirrungen geführt, da das Toolset dazu weiterhin unter dem Namen Swagger geführt wird. Eine Leseempfehlung für alle, die wissen wollen, wo genau die Unterschiede liegen und wie sich das Ganze entwickelt hat, gibt es unter [2]. So richtig durchgesetzt hat sich diese Namensänderung meiner Meinung nach nicht, meistens wird sowohl für die Spezifikation als auch für das Toolset nach wie vor der Begriff Swagger verwendet. Wie man es auch nennen mag – für uns hat sich diese Form sehr bewährt. Sowohl als Kommunikationshilfe bei der Definition eines API vor einem Projekt als auch zur Dokumentation für laufende Projekte hilft sie, Klarheit zu schaffen und Missverständnisse zu vermeiden.

Verwendung mit Symfony

Damit wir unsere Schnittstellenspezifikation nicht selbst schreiben müssen, verwenden wir in Kombination mit Symfony das NelmioApiDocBundle [3]. Das sucht sich u. a. aus Konfigurationsdateien, den konfigurierten Routen und speziellen Swagger Annotations Informationen zusammen und generiert daraus eine Spezifikation, die als JSON oder in einer WebView ausgespielt werden kann.

Das möchte ich euch gerne anhand eines Beispiels zeigen. Unsere Ausgangslage ist eine Installation von Symfony 4.3 mit einer einfachen Entity *Contact*, die folgende Properties hat: *id*, *name*, *email*, *country*. Dazu gibt es einen *ContactController*, der Actions für *GET*, *POST*, *PUT* und *DELETE* implementiert. Den vollständigen Code zu diesem Artikel findet ihr in meinem Repository [4]. Bei der Installation des NelmioApiDocBundle wird uns dank des zugehörigen Symfony Recipes bereits eine Konfigurationsdatei erstellt. In dieser

können wir u. a. den Titel und die Beschreibung unserer Schnittstellen angeben und vor allem auch festlegen, welche Routen in der Spezifikation vorkommen sollen (Listing 1).

Allein aus dieser Konfiguration und den schon vorhandenen Routenkonfigurationen kann das Bundle bereits eine minimale Spezifikation des bestehenden API erstellen. Im *ContactController* aus meinem Beispiel in Listing 1 ist in den Requirements der Routenkonfiguration für das GET API definiert, dass es einen Queryparameter *id* gibt und dieser nur aus Zahlen bestehen darf. Solche Vorgaben erkennt das Bundle automatisch. Unter dem Pfad */api/doc.json* ist diese minimale Spezifikation jetzt erreichbar.

Listing 1

```
# config/packages/nelmio_api_doc.yaml
nelmio_api_doc:
  documentation:
    info:
      title: Contacts API
      description: An API for fetching, adding, adapting and deleting contacts.
      version: 1.0.0
    areas: # Include everything available at /api except the doc APIs.
      default:
        path_patterns:
          - ^/api(?!/(doc.json|doc)$)
```

Anreicherung und Vervollständigung

Diese Spezifikation wollen wir jetzt vervollständigen. Dafür können eine Reihe von Annotations verwendet werden, die im Namespace *Swagger\Annotations* liegen und dazu dienen, Requests und mögliche Responses genauer zu definieren. Man startet mit der Annotation für die HTTP-Methode, z. B. *@Swagger\Annotations\Put* und übergibt dieser eine Summary, den Content-Typ der Response und alle Parameter und möglichen Responses. Bei den einzelnen Parametern kann genau definiert werden, wie sie aussehen müssen und was ein beispielhafter Wert wäre. Zum Beispiel

Listing 2

```
<?php
use Swagger\Annotations as SWG;
use Symfony\Component\Routing\Annotation\Route;
...

class ContactController
{
  ...
    /**
     * @Route("/api/contacts/{id}", name="contacts_put", methods={"PUT"},
                                                    requirements={"id"="\d+"})
     *
     * @SWG\Put(
     *     summary="Updates an existing contact.",
     *     produces={"application/json"},
     *     @SWG\Parameter(name="id", in="path", required=true, type="integer"),
     *     @SWG\Parameter(
     *         name="body",
     *         description="Post data.",
     *         in="body",
     *         required=true,
     *         @SWG\Schema(
     *             type="object",
     *             required={"name", "country"},
     *             @SWG\Property(
     *                 property="name",
     *                 type="string",
     *                 minLength=1,
     *                 example="Mia Muster"
     *             ),
     *             @SWG\Property(
     *                 property="email",
     *                 type="string",
     *                 format="email",
     *                 example="mia@muster.com"
     *             ),
     *             @SWG\Property(
     *                 property="country",
     *                 description="ISO-2 country code in capital letters.",
     *                 type="string",
     *                 pattern="^[A-Z]{2}$",
     *                 example="AT"
     *             )
     *         )
     *     ),
     *     @SWG\Response(
     *         response=200,
     *         description="Returns status 200 and the modified contact.",
     *         @SWG\Schema(
     *             type="object",
     *             properties={
     *                 @SWG\Property(property="id", type="integer"),
     *                 @SWG\Property(property="name", type="string"),
     *                 @SWG\Property(property="email", type="string"),
     *                 @SWG\Property(property="country", type="string")
     *             }
     *         )
     *     ),
     *     @SWG\Response(
     *         response=404,
     *         description="Returns status 404 if there is no contact with the given id."
     *     )
     * )
     */
    public function putAction(Request $request, int $id): JsonResponse
    {...}
}
```

wollen wir festlegen, dass die Property *country* aus genau zwei Buchstaben bestehen muss und ein Pflichtfeld ist. Listing 2 zeigt, wie die Annotations für unser PUT API aussehen könnten.

Wenn wir jetzt noch einmal */api/doc.json* aufrufen, erhalten wir eine vollständige Dokumentation unserer Schnittstelle (Listing 3). Das JSON wird dabei bei jedem Aufruf neu generiert, d. h., es muss kein Command ausgeführt werden, damit Änderungen der Spezifikation übernommen werden.

Swagger UI

Das Bundle kann uns aber nicht nur ein JSON ausspielen, sondern auch eine WebView, das sogenannte Swagger UI. Sobald man die entsprechende Route in der Datei *config/routes/nelmio_api_doc.yaml* angepasst hat, ist diese unter */api/doc* erreichbar (**Abb. 1**). Mit einem Klick auf eine einzelne Schnittstelle öffnet sich die Detailspezifikation und ein Button TRY IT OUT wird sichtbar. Mit diesem können Schnittstellen direkt aus der Oberfläche heraus aufgerufen und getestet werden.

Validierung

Für diese Spezifikation war jetzt schon einiges an Schreibarbeit notwendig und wir wollen das auf keinen Fall noch einmal alles wiederholen müssen, um Daten, die über *POST* oder *PUT* hereinkommen, zu validie-

Listing 3

```
{
 "swagger": "2.0",
 "info": {
  "title": "Contacts API",
  "description": "An API for fetching, adding, adapting and deleting contacts.",
  "version": "1.0.0"
 },
 "paths": {
  "/api/contacts/{id}": {
   "get": {...},
   "put": {
    "summary": "Updates an existing contact.",
    "produces": ["application/json"],
    "parameters": [
     {
      "name": "id",
      "in": "path",
      "required": true,
      "type": "integer",
      "pattern": "\\d+"
     },
     {
      "name": "body",
      "in": "body",
      "required": true,
      "description": "Post data.",
      "schema": {
       "required": [
        "name",
        "country"
       ],
       "properties": {
        "name": {
         "example": "Mia Muster",
         "type": "string",
         "minLength": 1
        },
        "email": {
         "example": "mia@muster.com",
         "type": "string",
         "format": "email"
        },
        "country": {
         "description": "ISO-2 country code in capital letters.",
         "example": "AT",
         "type": "string",
         "pattern": "^[A-Z]{2}$"
        }
       },
       "type": "object"
      }
     }
    ],
    "responses": {
     "200": {
      "description": "Returns status 200 and the modified contact.",
      "schema": {
       "properties": {
        "id": {"type": "integer"},
        "name": {"type": "string"},
        "email": {"type": "string"},
        "country": {"type": "string"}
       },
       "type": "object"
      }
     },
     "404": {
      "description": "Returns status 404 if there is no contact with the given id."
     }
    }
   },
   "delete": {...}
  },
  "/api/contacts": {
   "post": {...}
  }
 }
}
```

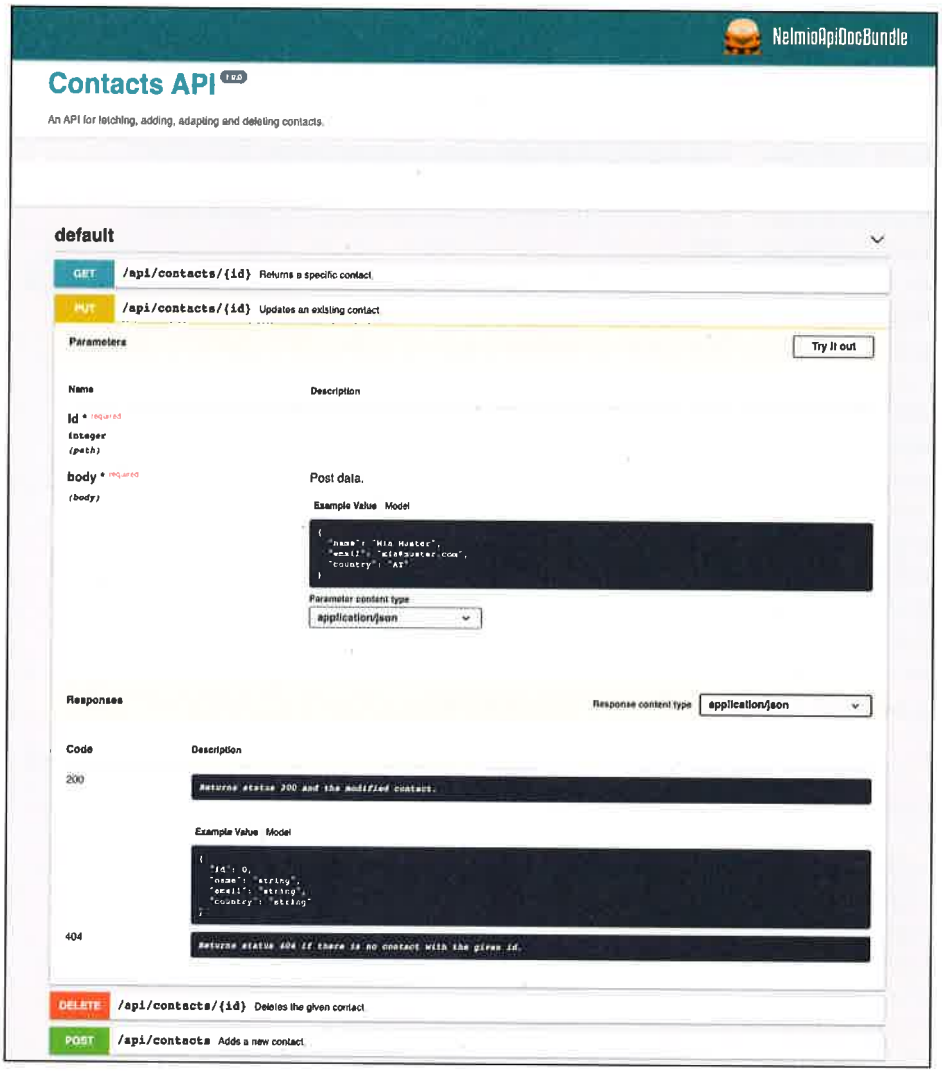

Abb. 1: Swagger UI

ren. Darum verwenden wir zusätzlich noch das Linkin-SwaggerResolverBundle [5]. Es validiert die Daten aus dem Request gegen unsere Spezifikation und wirft verschiedene Exceptions, falls etwas nicht passt. Die Exceptions implementieren alle das *ExceptionInterface* aus dem OptionsResolver-Bundle, d. h., es ist einfach, später genau für diese Exceptions ein Fehlerhandling einzubauen. Wir erweitern unseren Controller um eine Methode *validateRequest()*, die die Factory aus diesem Bundle verwendet und die Request-Daten aus URL und Body übergibt (Listing 4). Diese Methode ist ganz allgemein gehalten und könnte zur Wiederverwendung in einen Trait oder einen abstrakten *ApiController* ausgelagert werden.

Das Bundle meldet uns in Form der Exception-Message sogar detailliert in Textform zurück, wieso ein Request fehlgeschlagen ist. Mit Hilfe eines *ExceptionListener*, der auf das Symfony Event *onKernelException* hört, können wir diese Exceptions abfangen und die Information zur fehlgeschlagenen Validierung als JSON-Response zurückgeben (Listing 5). Wird z. B. als *country* ein dreistelliger String an das API gesendet, gibt unser Listener automatisch Status 400 mit der Message-Property *"country" should match the pattern "/^[A-Z]{2}$/"* zurück.

Deserializing und Serializing

Jetzt, wo sichergestellt ist, dass die Daten valide sind, können wir sie zur weiteren Verwendung umwandeln – in unserem Fall in ein *Contact*-Objekt. Und später wollen wir umgekehrt das Objekt in ein JSON umwandeln, um es als Response zurückzuschicken. Anstatt diese Umwandlungen manuell z. B. durch Zuweisungen der Werte zu den einzelnen Properties des Objekts zu machen, kann ein sogenannter Serializer verwendet werden. Für unsere einfache Entity schafft das der Symfony Serializer ohne weitere Konfiguration oder Angabe von Informationen. Listing 6 zeigt

Listing 4

```
<?php
use Linkin\Bundle\SwaggerResolverBundle\Factory\SwaggerResolverFactory;
...

class ContactController
{
...
 public function putAction(Request $request, int $id): JsonResponse
 {
  $this->validateRequest($request);
  ...
 }

 private function validateRequest(Request $request)
 {
  $swaggerResolver = $this->swaggerResolverFactory->createForRequest($request);
  $swaggerResolver->resolve(array_merge(
   json_decode($request->getContent(), true),
   $request->attributes->get('_route_params')
  ));
 }
}
```

die Verwendung des Serializers mit den Methoden *updateObject()* und *createResponse()*. Die Methoden sind sehr allgemein gehalten und könnten nicht nur für unser *Contact*-Objekt, sondern für jedes beliebige serialisierbare Objekt verwendet werden.

Werden die Objekte komplexer, ist das nicht mehr ganz so einfach. Der Serializer bietet aber verschiedene Möglichkeiten, genauer festzulegen, welche Properties wie serialisiert oder welche z. B. komplett ausgeschlossen werden sollen. Mit Hilfe von Serialization Groups kann das gleiche Objekt für verschiedene Schnittstellen verschieden serialisiert werden. Wenn ich in einer Kontaktübersicht nur den Namen anzeigen will und die restlichen Informationen ausschließlich für eine Detailansicht brauche, kann ich zwei Gruppen definieren, die genau die gewünschten Properties enthalten. Beim Serialisieren muss dann nur der Name der gewünschten Gruppe angegeben werden. Ehrlicherweise muss man aber sagen, dass das bei komplexen Objekten mit Subobjekten zu einer Herausforderung werden kann, wenn es z. B. darum geht, endlose Schleifen bei Many-to-Many-Relationen zu verhindern.

Fazit

Dieses Set-up zeigt, wie verschiedene Teile einer Schnittstellenimplementierung generalisiert werden können und macht es einem einfach, neue APIs zu implementieren. Es bietet ein einheitliches Handling von Validierungsfehlern für alle Schnittstellen, und die unter Entwicklern so unbeliebte Dokumentation kann mit Hilfe der richtigen Annotations automatisch generiert werden. Wir können leider nicht alle Duplikate vermeiden, aber einige Wiederholungen kann man sich auf jeden Fall sparen.

Listing 5

```
<?php
use Symfony\Component\OptionsResolver\Exception\ExceptionInterface;
...

class ExceptionListener
{
  public function onKernelException(ExceptionEvent $event)
  {
    $exception = $event->getException();

    // Catch exceptions from the validator.
    if ($exception instanceof ExceptionInterface) {
      $response = new JsonResponse([
        'status' => Response::HTTP_BAD_REQUEST,
        'message' => $exception->getMessage()
      ], Response::HTTP_BAD_REQUEST);

      $event->setResponse($response);
    }
  }
}
```

Sabine Bär hat ihre Begeisterung für Webentwicklung schon früh entdeckt und diese Leidenschaft nach ihrem Informatikstudium zum Beruf gemacht. Als Softwarearchitektin bei MASSIVE ART Web-Services GmbH in Dornbirn (AT) trägt sie maßgeblich zum Erfolg innovativer Webprojekte bei. Fast ebenso viel Begeisterung hegt sie für die Musik – sowohl passiv als auch aktiv beim Musikverein.

Links & Literatur

[1] https://www.openapis.org
[2] https://swagger.io/blog/api-strategy/difference-between-swagger-and-openapi/
[3] https://github.com/nelmio/NelmioApiDocBundle
[4] https://github.com/sabinebaer/symfony-apis
[5] https://github.com/adrenalinkin/swagger-resolver-bundle

Listing 6

```
<?php
use Symfony\Component\Serializer\SerializerInterface;

class ContactController
{
  ...
  public function putAction(Request $request, int $id): JsonResponse
  {
    $this->validateRequest($request);
    $contact = $this->contactRepository->find($id);
    ...

    $this->updateObject($contact, $request->getContent());
    ...

    return $this->createJsonResponse($contact);
  }

  private function updateObject(object $target, string $jsonData)
  {
    $this->serializer->deserialize(
      $jsonData,
      get_class($target),
      'json',
      ['object_to_populate' => $target]
    );
  }

  private function createJsonResponse(object $data): JsonResponse
  {
    $jsonData = $this->serializer->serialize($data, 'json');

    return new JsonResponse($jsonData, Response::HTTP_OK, [], true);
  }
  ...
}
```