6 REST-Dienste erstellen

Die Themen dieses Kapitels:

- REST-Endpunkte in Spring MVC definieren
- Per Hyperlink verknüpfte REST-Ressourcen aktivieren
- Automatische Repository-basierte REST-Endpunkte

»Der Webbrowser ist tot. Was nun?«

Vor etwa einem Dutzend Jahren ist mir die Behauptung zu Ohren gekommen, dass sich der Webbrowser dem Legacy-Status nähert und dass etwas anderes an seine Stelle treten würde. Doch wie kann das sein? Was könnte den nahezu allgegenwärtigen Webbrowser vom Thron stoßen? Wie könnten wir die wachsende Anzahl von Sites und Onlinediensten konsumieren, wenn nicht mit einem Webbrowser? Sicherlich war das nur das Geschwafel eines Verwirrten!

Ein Sprung in die Gegenwart macht deutlich, dass der Webbrowser nicht verschwunden ist. Aber er regiert nicht mehr als primäres Mittel für den Zugang zum Internet. Mobile Geräte, Tablets, Smartwatches und sprachgesteuerte Geräte sind heute an der Tagesordnung. Und selbst viele browserbasierte Anwendungen führen eigentlich JavaScript-Anwendungen aus, anstatt den Browser zu einem dummen Terminal für die vom Server gerenderten Inhalte zu machen.

Bei einer so riesigen Auswahl an clientseitigen Optionen haben viele Anwendungen ein gemeinsames Design angenommen, bei dem die Benutzeroberfläche näher an den Client gerückt wird und der Server eine API bereitstellt, über die sämtliche Arten von Clients mit der Backend-Funktionalität interagieren können.

In diesem Kapitel erstellen Sie mit Spring eine REST API für die Taco-Cloud-Anwendung. Dabei bauen Sie auf dem auf, was Sie in Kapitel 2 über Spring MVC gelernt haben, um RESTful-Endpunkte mit Spring MVC-Controllern zu erstellen. Außerdem machen Sie REST-Endpunkte für die Spring-Data-Repositories zugänglich, die Sie in Kapitel 4 definiert haben. Schließlich sehen wir uns Möglichkeiten an, diese Endpunkte zu testen und zu sichern.

Aber zuerst programmieren Sie ein paar neue Spring-MVC-Controller, die Backend-Funktionalität mit REST-Endpunkten zugänglich machen, sodass ein Rich-Web-Frontend sie konsumieren kann.

6.1 RESTful Controller programmieren

Ich hoffe, es macht Ihnen nichts aus, doch Sie werden feststellen, dass ich die Benutzeroberfläche für Taco Cloud neu gestaltet habe. Womit Sie bisher gearbeitet haben, war für den Anfang gewiss in Ordnung, doch es hapert an der Ästhetik.

Bild 6.1 ist nur ein Beispiel dafür, wie die neue Taco-Cloud aussieht. Ganz pfiffig, oder?

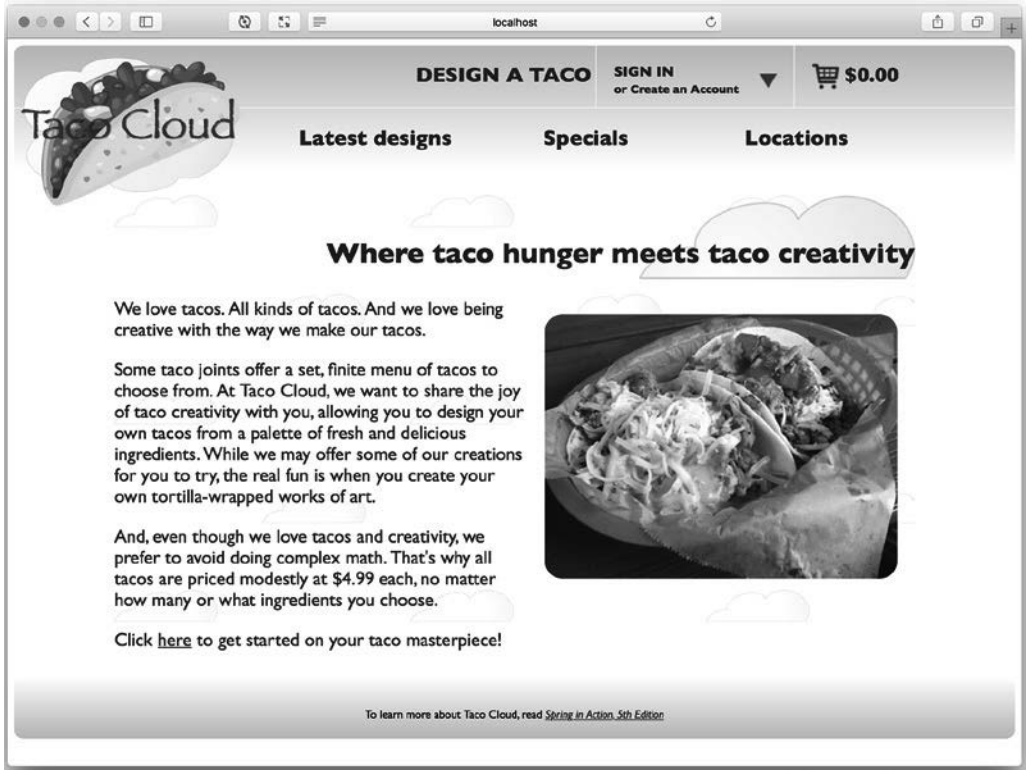

Bild 6.1 Die neue Taco-Cloud-Startseite

Und da ich schon dabei war, den Taco-Cloud-Look aufzupolieren, habe ich mich auch entschlossen, den Frontend als Single-Page-Anwendung mit dem beliebten Angular-Framework zu erstellen. Schließlich wird diese neue Browser-Benutzeroberfläche die vom Server gerenderten Seiten ersetzen, die Sie in Kapitel 2 erstellt haben. Doch damit das funktioniert, müssen Sie eine REST API erstellen, mit der die Angular-basierte[1] Benutzeroberfläche kommuniziert, um Taco-Daten zu speichern und abzurufen.

[1] Ich setze hier auf Angular, doch die Wahl des Frontend-Frameworks sollte wenig oder gar keinen Einfluss darauf haben, wie der Backend-Spring-Code entsteht. Entscheiden Sie sich für die Frontend-Technologie, die Ihnen am meisten zusagt – Angular, React, Vue.js oder eine andere.

SPA oder nicht SPA?

In Kapitel 2 haben Sie mit Spring MVC eine herkömmlich mehrseitige Anwendung (Multi-Page-Application, MPA) entwickelt. Jetzt ersetzen Sie diese durch eine Einzelseiten-Anwendung (Single-Page-Application, SPA) basierend auf Angular. Doch ich behaupte nicht, dass SPA immer die bessere Wahl als MPA ist.

Da die Präsentation in einer SPA weitgehend von der Backend-Verarbeitung entkoppelt ist, bietet sie die Möglichkeit, mehr als eine Benutzeroberfläche (beispielsweise eine native mobile Anwendung) für dieselbe Backend-Funktionalität zu entwickeln. Auch die Integration mit anderen Anwendungen, die die API konsumieren können, ist damit möglich. Allerdings brauchen nicht alle Anwendungen diese Flexibilität und das Design einer MPA ist einfacher, wenn Sie lediglich Informationen auf einer Webseite anzeigen müssen.

Dies ist kein Buch über Angular und somit geht es beim Code in diesem Kapitel vorrangig um den Backend-Spring-Code. Angular-Code gebe ich nur insoweit an, dass Sie ein Gefühl dafür bekommen, wie die Clientseite funktioniert. Auf jeden Fall können Sie den vollständigen Code, einschließlich des Angular-Frontends, von der Webseite unter https://github.com/habuma/spring-in-action-5-samples herunterladen.

Kurz gesagt kommuniziert der Angular-Clientcode über HTTP-Anfragen mit einer API, die Sie im Verlauf dieses Kapitels erstellen. In Kapitel 2 haben Sie die Annotationen `@GetMapping` und `@PostMapping` verwendet, um Daten vom Server abzurufen und an den Server zu senden. Die gleichen Annotationen erweisen sich auch als nützlich, wenn Sie Ihre REST API definieren. Darüber hinaus unterstützt Spring MVC eine Handvoll anderer Annotationen für verschiedene Typen von HTTP-Anfragen. Diese sind in Tabelle 6.1 aufgelistet.

Tabelle 6.1 Annotationen von Spring MVC für die Behandlung von HTTP-Anfragen

Annotation	HTTP-Methode	Typische Verwendung*
`@GetMapping`	HTTP-`GET`-Anfragen	Ressourcendaten lesen
`@PostMapping`	HTTP-`POST`-Anfragen	Eine Ressource erstellen
`@PutMapping`	HTTP-`PUT`-Anfragen	Eine Ressource aktualisieren
`@PatchMapping`	HTTP-`PATCH`-Anfragen	Eine Ressource aktualisieren
`@DeleteMapping`	HTTP-`DELETE`-Anfragen	Eine Ressource löschen
`@RequestMapping`	Universelle Anfragebehandlung; HTTP-Methode im `method`-Attribut angegeben	

* Die Zuordnung der HTTP-Methoden zu CRUD- (Create, Read, Update, Delete)-Operationen ist nicht 100%ig perfekt, wird aber in der Praxis meistens so gehandhabt und gilt auch für die Verwendung in Taco Cloud.

Um diese Annotationen in Aktion zu sehen, erstellen Sie zunächst einen einfachen REST-Endpunkt, der ein paar der zuletzt kreierten Tacos abruft.

6.1.1 Daten vom Server abrufen

Eines der coolsten Features der Taco-Cloud-Anwendung ist es, dass Taco-Fans ihre eigenen Taco-Kreationen schaffen und sie mit anderen Taco-Liebhabern teilen können. Zu diesem Zweck muss Taco Cloud in der Lage sein, eine Liste der neuesten Taco-Kreationen anzuzeigen, wenn der Link LATEST DESIGNS (»Neueste Kreationen«) angeklickt wird.

Im Angular-Code habe ich eine Klasse `RecentTacosComponent` definiert, die die zuletzt kreierten Tacos anzeigt. Der vollständige TypeScript-Code für `RecentTacosComponent` ist in Listing 6.1 zu sehen.

Listing 6.1 Angular-Komponente für die Anzeige der neuesten Tacos

```
import { Component, OnInit, Injectable } from '@angular/core';
import { Http } from '@angular/http';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'recent-tacos',
  templateUrl: 'recents.component.html',
  styleUrls: ['./recents.component.css']
})

@Injectable()
export class RecentTacosComponent implements OnInit {
  recentTacos: any;

  constructor(private httpClient: HttpClient) { }

  ngOnInit() {    ◂ Ruft neueste Tacos vom Server ab
    this.httpClient.get('http://localhost:8080/design/recent')
        .subscribe(data => this.recentTacos = data);
  }
}
```

Sehen Sie sich die Methode `ngOnInit()` an. In dieser Methode verwendet `RecentTacosComponent` das injizierte `Http`-Modul, um eine HTTP-`GET`-Anfrage an http://localhost:8080/design/recent auszuführen. Von der Antwort wird erwartet, dass sie eine Liste von Taco-Kreationen enthält, die in die Modellvariable `recentTacos` übernommen wird. Die View (in *recents.component.html*) präsentiert diese Modelldaten als HTML, das im Browser gerendert wird. Das Endergebnis sieht zum Beispiel wie in Bild 6.2 gezeigt aus, nachdem drei Tacos kreiert wurden.

Das fehlende Teil in diesem Puzzle ist ein Endpunkt, der `GET`-Anfragen für */design/recent* verarbeitet und mit einer Liste der zuletzt kreierten Tacos antwortet. Erstellen Sie also einen neuen Controller, um eine derartige Anfrage zu behandeln. Listing 6.2 zeigt den Controller für diese Aufgabe.

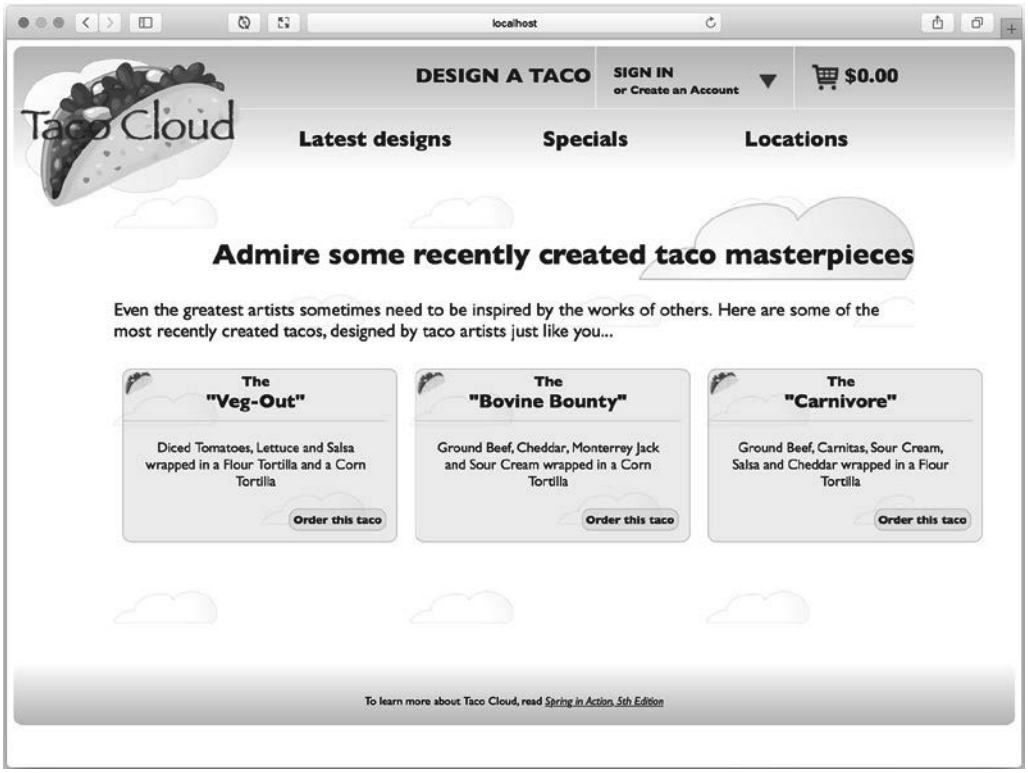

Bild 6.2 Die neuesten Taco-Kreationen anzeigen

Listing 6.2 Ein RESTful Controller für API-Anfragen nach Taco-Kreationen

```
package tacos.web.api;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.hateoas.EntityLinks;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import tacos.Taco;
import tacos.data.TacoRepository;

@RestController
@RequestMapping(path="/design",    ◂ Behandelt Anfragen nach /design
                produces="application/json")
```

```
@CrossOrigin(origins="*")    ◂ Erlaubt ursprungsübergreifende Anfragen
public class DesignTacoController {
  private TacoRepository tacoRepo;

  @Autowired
  EntityLinks entityLinks;

  public DesignTacoController(TacoRepository tacoRepo) {
    this.tacoRepo = tacoRepo;
  }

  @GetMapping("/recent")                    Ruft die neuesten Taco-Kreationen ab
  public Iterable<Taco> recentTacos() {    ◂ und gibt sie zurück
    PageRequest page = PageRequest.of(
            0, 12, Sort.by("createdAt").descending());
    return tacoRepo.findAll(page).getContent();
  }
}
```

Hinweis

In Listing 6.2 übernimmt die Methode `findAll()`, die auf `TacoRepository` aufgerufen wird, ein `PageRequest`-Objekt, um das Paging zu realisieren. Um diese spezielle `findAll()`-Methode verfügbar zu machen, wird `TacoRepository` so geändert, dass es `PagingAndSortingRepository` statt `CrudRepository` erweitert. Diese Änderung ist im herunterladbaren Beispielcode bereits enthalten, wird aber im Textteil des Buchs nirgends explizit erwähnt.

Der Name dieses Controllers mag Ihnen bekannt vorkommen. In Kapitel 2 haben Sie einen `DesignTacoController` erstellt, der ähnliche Anforderungen verarbeitet hat. Allerdings war dieser Controller auf die mehrseitige Taco-Cloud-Anwendung zugeschnitten, während der neue `DesignTacoController` ein REST-Controller ist, wie es die Annotation `@RestController` anzeigt.

Die Annotation `@RestController` erfüllt zwei Aufgaben. Erstens ist es eine stereotype Annotation wie `@Controller` und `@Service`, die eine Klasse für die Erkennung bei der Komponentensuche markiert. Vor allem aber ist es im Zusammenhang mit REST wichtig, dass die Annotation `@RestController` Spring sagt, dass alle Behandlungsmethoden im Controller ihren Rückgabewert direkt in den Rumpf der Antwort schreiben sollen, anstatt ihn im Modell an eine View zum Rendern zu übertragen.

Alternativ könnten Sie auch `DesignTacoController` mit `@Controller` annotieren, genau wie bei jedem Spring-MVC-Controller. Doch dann müssten Sie auch alle Behandlungsmethoden mit `@ResponseBody` annotieren, um das gleiche Ergebnis zu erhalten. Noch eine andere Option wäre es, ein `ResponseEntity`-Objekt zurückzugeben, worauf wir in Kürze eingehen.

Die Annotation `@RequestMapping` auf der Klassenebene spezifiziert zusammen mit der Annotation `@GetMapping` auf der Methode `recentTacos()`, dass die Methode `recentTacos()` dafür zuständig ist, `GET`-Anfragen für */design/recent* zu verarbeiten (was genau das ist, was Ihr Angular-Code benötigt).

Beachten Sie, dass die Annotation `@RequestMapping` auch ein Attribut `produces` festlegt. Es gibt an, dass die Behandlungsmethoden in `DesignTacoController` Anfragen nur verarbeiten, wenn der `Accept`-Header der Anfrage den String `"application/json"` enthält. Dies schränkt Ihre API nicht nur dahingehend ein, ausschließlich JSON-Ergebnisse zu produzieren, es sorgt auch dafür, dass ein anderer Controller (vielleicht der `DesignTacoController` aus Kapitel 2) Anfragen mit den gleichen Pfaden verarbeiten kann, solange diese Anfragen keine JSON-Ausgabe verlangen. Auch wenn Ihre API dadurch JSON-basiert sein muss (was für Ihre Zwecke in Ordnung ist), können Sie gern auch `produces` auf ein Array von `String`-Werten für mehrere Inhaltstypen setzen. Möchten Sie beispielsweise XML-Ausgaben erlauben, fügen Sie dem `produces`-Attribut `"text/html"` hinzu:

```
@RequestMapping(path="/design",
                produces={"application/json", "text/xml"})
```

In Listing 6.2 ist Ihnen sicherlich auch aufgefallen, dass die Klasse mit `@CrossOrigin` annotiert ist. Da der Angular-Teil der Anwendung auf einem separaten Host und/oder Port von der API ausgeführt wird (zumindest vorerst), hindert der Webbrowser Ihren Angular-Client daran, die API zu konsumieren. Diese Einschränkung lässt sich überwinden, indem Sie CORS (Cross-Origin Resource Sharing)-Header in die Server-Antworten aufnehmen. In Spring ist es mit der Annotation `@CrossOrigin` leicht, CORS anzuwenden. So wie es hier verwendet wird, erlaubt `@CrossOrigin` Clients aus jeder Domäne, die API zu verwenden.

Die Logik in der Methode `recentTacos()` ist ziemlich einfach. Die Methode konstruiert ein `PageRequest`-Objekt, das festlegt, dass Sie nur die erste (0-te) Seite von zwölf Ergebnissen haben möchten, und zwar absteigend sortiert nach dem Erstellungsdatum des Tacos. Kurz gesagt möchten Sie ein Dutzend der neuesten Taco-Kreationen abrufen. Das `PageRequest`-Objekt wird im Aufruf der Methode `findAll()` des `TacoRepository` übergeben und der Inhalt dieser Ergebnisseite wird an den Client zurückgegeben (der, wie Listing 6.1 gezeigt hat, als Modelldaten zur Anzeige für den Benutzer verwendet wird).

Nehmen wir nun an, dass Sie einen Endpunkt anbieten möchten, der einen einzelnen Taco nach seiner ID abruft. Indem Sie eine Platzhaltervariable im Pfad der Behandlungsmethode verwenden und eine Pfadvariable übernehmen, können Sie die ID erfassen und für die Suche nach dem `Taco`-Objekt im Repository heranziehen:

```
@GetMapping("/{id}")
public Taco tacoById(@PathVariable("id") Long id) {
  Optional<Taco> optTaco = tacoRepo.findById(id);
  if (optTaco.isPresent()) {
    return optTaco.get();
  }
  return null;
}
```

Da der Basispfad des Controllers */design* lautet, verarbeitet diese Controller-Methode `GET`-Anfragen für `/design/{id}`, wobei der `{id}`-Teil des Pfads ein Platzhalter ist. Der tatsächliche Wert in der Anfrage wird an den `id`-Parameter übergeben, der durch `@PathVariable` auf den Platzhalter `{id}` abgebildet wird.

In der Methode `tacoById()` wird der Parameter `id` an die Methode `findById()` übergeben, um das `Taco`-Objekt abzurufen. Die Methode `findById()` gibt ein `Optional<Taco>` zurück, da möglicherweise kein Taco mit der angegebenen ID existiert. Deshalb müssen Sie ermitteln, ob die ID mit einem Taco übereinstimmt oder nicht, bevor ein Wert zurückgegeben wird. Stimmt die ID überein, rufen Sie `get()` auf dem `Optional<Taco>`-Objekt auf, um das eigentliche `Taco`-Objekt zurückzugeben.

Wenn die ID keinem der bekannten Tacos entspricht, geben Sie `null` zurück. Allerdings ist das nicht ideal, denn der Client empfängt dann eine Antwort mit einem leeren Rumpf, aber mit einem HTTP-Statuscode von 200 (OK). Der Client erhält eine Antwort, mit der er nichts anfangen kann, doch der Statuscode besagt, dass alles in Ordnung ist. Eine bessere Lösung wäre es, eine Antwort mit einem HTTP-Statuscode 404 (Nicht gefunden) zurückzugeben.

Im derzeitigen Code gibt es aber keine einfache Möglichkeit, einen Statuscode 404 aus `tacoById()` zurückzugeben. Doch mit einigen Anpassungen können Sie den Statuscode ordnungsgemäß festlegen:

```
@GetMapping("/{id}")
public ResponseEntity<Taco> tacoById(@PathVariable("id") Long id) {
  Optional<Taco> optTaco = tacoRepo.findById(id);
  if (optTaco.isPresent()) {
    return new ResponseEntity<>(optTaco.get(), HttpStatus.OK);
  }
  return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
}
```

Anstatt nun ein `Taco`-Objekt zurückzugeben, gibt `tacoById()` ein `ResponseEntity<Taco>` zurück. Wird der Taco gefunden, hüllen Sie das `Taco`-Objekt in eine `ResponseEntity` mit dem HTTP-Status OK (was das gleiche Verhalten ergibt wie zuvor). Doch wenn der Taco nicht gefunden wird, hüllen Sie eine `null` zusammen mit dem HTTP-Status NOT FOUND in ein `ResponseEntity`-Objekt und zeigen damit an, dass der Client versucht, einen Taco abzurufen, der nicht existiert.

Damit haben Sie den Grundstein einer Taco-Cloud-API für Ihren Angular-Client gelegt – oder sogar für jede andere Art von Client. Für das Testen in der Entwicklungsphase bieten sich auch die Befehlszeilenprogramme wie `curl` oder `HTTPie` (https://httpie.org/) an, um in der API herumzustöbern. So zeigt die folgende Befehlszeile ein Beispiel, wie Sie die neuesten Taco-Kreationen mit `curl` abrufen:

```
$ curl localhost:8080/design/recent
```

Wenn Sie `HTTPie` vorziehen, sieht der Befehl so aus:

```
$ http :8080/design/recent
```

Einen Endpunkt zu definieren, der Informationen zurückgibt, ist aber nur der Anfang. Wie sieht es aus, wenn Ihre API Daten vom Client empfangen muss? Der nächste Abschnitt zeigt, wie Sie Controller-Methoden schreiben, die Eingaben in den Anfragen verarbeiten.

6.1.2 Daten an den Server senden

Ihre API ist nun in der Lage, ein Dutzend der neuesten Taco-Kreationen zurückzugeben. Doch wie werden diese Tacos überhaupt erzeugt?

Da Sie noch keinen Code aus Kapitel 2 gelöscht haben, gibt es immer noch den ursprünglichen `DesignTacoController`, der ein Taco-Design-Formular anzeigt und die Formularübermittlung behandelt. Das ist eine großartige Möglichkeit, einige Testdaten zu bekommen, um die eben erstellte API zu testen. Doch wenn Sie Taco Cloud in eine Einzelseiten-Anwendung umwandeln, müssen Sie Angular-Komponenten und entsprechende Endpunkte erzeugen, um dieses Taco-Design-Formular aus Kapitel 2 zu ersetzen.

Der Clientcode für das Taco-Design-Formular ist bereits bearbeitet, und zwar habe ich eine neue Angular-Komponente namens `DesignComponent` (in einer Datei *design.component.ts*) definiert. Für die Verarbeitung der Formularübermittlung besitzt `DesignComponent` eine Methode `onSubmit()`, die folgendermaßen aussieht:

```
onSubmit() {
  this.httpClient.post(
      'http://localhost:8080/design',
      this.model, {
          headers: new HttpHeaders().set('Content-type', 'application/json'),
      }).subscribe(taco => this.cart.addToCart(taco));

  this.router.navigate(['/cart']);
}
```

Die Methode `onSubmit()` ruft die `HttpClient`-Methode `post()` statt `get()` auf. Anstatt also Daten aus der API abzurufen, senden Sie Daten an die API. Insbesondere senden Sie eine Taco-Kreation, die in der Variablen `model` gespeichert ist, mit einer HTTP-POST-Anfrage an den API-Endpunkt unter */design*.

Deshalb brauchen Sie in `DesignTacoController` eine Methode, um diese Anfrage zu verarbeiten und die Kreation zu speichern. Indem Sie die folgende Methode `postTaco()` zu `DesignTacoController` hinzufügen, versetzen Sie den Controller in die Lage, genau das zu tun:

```
@PostMapping(consumes="application/json")
@ResponseStatus(HttpStatus.CREATED)
public Taco postTaco(@RequestBody Taco taco) {
  return tacoRepo.save(taco);
}
```

Da `postTaco()` eine HTTP-POST-Anfrage verarbeitet, wird die Methode mit `@PostMapping` statt `@GetMapping` annotiert. Hier ist kein `path`-Attribut angegeben und so verarbeitet die Methode `postTaco()` Anfragen für */design*, wie es in der Annotation `@RequestMapping` auf Klassenebene bei `DesignTacoController` festgelegt ist.

Allerdings setzen Sie das Attribut `consumes`. Es wirkt bei Eingaben für Anfragen wie `produces` für die Ausgabe einer Anfrage. Hier drücken Sie mit `consumes` aus, dass die Methode nur Anfragen verarbeitet, deren `Content-type` mit `application/json` übereinstimmt.

Die Annotation `@RequestBody` am Parameter `Taco` zeigt an, dass der Rumpf der Anfrage in ein `Taco`-Objekt konvertiert und an den Parameter gebunden werden sollte. Diese Annotation ist wichtig – ohne sie würde Spring MVC annehmen, dass Sie Anfrageparameter (entweder Parameter einer Abfrage oder Formularparameter) an das `Taco`-Objekt binden möchten. Aber die Annotation `@RequestBody` stellt sicher, dass stattdessen das JSON im Anfragerumpf an das `Taco`-Objekt gebunden wird.

Sobald `postTaco()` das `Taco`-Objekt empfangen hat, wird es an die Methode `save()` im `TacoRepository` übergeben.

Vielleicht haben Sie auch bemerkt, dass ich die Methode `postTaco()` mit `@ResponseStatus(HttpStatus.CREATED)` annotiert habe. Unter normalen Umständen (wenn keine Ausnahmen ausgelöst werden), zeigen alle Antworten mit dem HTTP-Statuscode 200 (OK) an, dass die Anfrage erfolgreich war. Eine Antwort mit dem HTTP-Statuscode 200 ist zwar stets willkommen, aber nicht immer aussagekräftig genug. Bei einer `POST`-Anfrage ist ein HTTP-Status von `201 (CREATED)` deskriptiver. Der Client erkennt daran, dass nicht nur die Anfrage erfolgreich war, sondern im Ergebnis auch eine Ressource erzeugt wurde. Wenn möglich sollten Sie `@ResponseStatus` verwenden, um dem Client den aussagekräftigsten und genauesten HTTP-Statuscode zu kommunizieren.

Obwohl Sie mit `@PostMapping` eigentlich eine neue `Taco`-Ressource erzeugen, lassen sich `POST`-Anfragen auch nutzen, um Ressourcen zu aktualisieren. Dennoch verwendet man `POST`-Anfragen in der Regel für das Erstellen von Ressourcen und `PUT`- und `PATCH`-Anfragen für das Aktualisieren der Ressourcen. Sehen Sie sich nun an, wie Sie Daten mittels `@PutMapping` und `@PatchMapping` aktualisieren können.

6.1.3 Daten auf dem Server aktualisieren

Bevor Sie irgendwelchen Controllercode für die Verarbeitung von HTTP-PUT- oder PATCH-Befehlen schreiben, sollten Sie sich etwas Zeit nehmen und erst einmal den Elefanten im Raum betrachten: Warum gibt es zwei verschiedene HTTP-Methoden, um Ressourcen zu aktualisieren?

Es stimmt zwar, dass `PUT` oftmals verwendet wird, um Ressourcendaten zu aktualisieren, doch ist es eigentlich das semantische Gegenstück zu `GET`. Während `GET`-Anfragen dafür gedacht sind, Daten vom Server zum Client zu übertragen, sollen `PUT`-Anfragen Daten vom Client zum Server senden.

In diesem Sinne ist `PUT` wirklich darauf ausgerichtet, eine Ersetzungsoperation en gros statt einer Aktualisierung durchzuführen. Im Gegensatz dazu soll HTTP `PATCH` einen Patch oder eine teilweise Aktualisierung von Ressourcendaten durchführen.

Angenommen, Sie möchten die Adresse in einer Bestellung ändern können. Dies ließe sich zum Beispiel über die REST API mit einer `PUT`-Anfrage bewerkstelligen, die wie folgt behandelt wird:

```
@PutMapping("/{orderId}")
public Order putOrder(@RequestBody Order order) {
  return repo.save(order);
}
```

Dies könnte funktionieren, setzt aber voraus, dass der Client die vollständigen Bestelldaten in der PUT-Anfrage übermittelt. Semantisch heißt PUT: »Stelle diese Daten an diese URL«, wobei praktisch alle bereits dort befindlichen Daten ersetzt werden. Fehlt eine Eigenschaft der Bestellung, wird der Wert der Eigenschaft mit `null` überschrieben. Sogar die Tacos in der Bestellung müssten Sie zusammen mit den Bestelldaten setzen, da sie sonst aus der Bestellung entfernt würden.

Wenn `PUT` die Ressourcendaten en gros ersetzt, wie sollten Sie dann Anfragen verarbeiten, um nur eine teilweise Aktualisierung zu erreichen? Dafür sind HTTP-`PATCH`-Anfragen und `@PatchMapping` von Spring geeignet. Der folgende Code zeigt eine Controllermethode, die eine `PATCH`-Anfrage für eine Bestellung verarbeitet:

```
@PatchMapping(path="/{orderId}", consumes="application/json")
public Order patchOrder(@PathVariable("orderId") Long orderId,
                        @RequestBody Order patch) {

  Order order = repo.findById(orderId).get();
  if (patch.getDeliveryName() != null) {
    order.setDeliveryName(patch.getDeliveryName());
  }
  if (patch.getDeliveryStreet() != null) {
    order.setDeliveryStreet(patch.getDeliveryStreet());
  }
  if (patch.getDeliveryCity() != null) {
    order.setDeliveryCity(patch.getDeliveryCity());
  }
  if (patch.getDeliveryState() != null) {
    order.setDeliveryState(patch.getDeliveryState());
  }
  if (patch.getDeliveryZip() != null) {
    order.setDeliveryZip(patch.getDeliveryState());
  }
  if (patch.getCcNumber() != null) {
    order.setCcNumber(patch.getCcNumber());
  }
  if (patch.getCcExpiration() != null) {
    order.setCcExpiration(patch.getCcExpiration());
  }
  if (patch.getCcCVV() != null) {
    order.setCcCVV(patch.getCcCVV());
  }

  return repo.save(order);
}
```

Hier ist zunächst festzustellen, dass die Methode `patchOrder()` mit `@PatchMapping` anstelle von `@PutMapping` annotiert ist, was anzeigt, dass die Methode HTTP-`PATCH`-Anfragen statt `PUT`-Anfragen verarbeiten soll.

Zweifellos haben Sie auch bemerkt, dass die Methode `patchOrder()` ein ganzes Stück komplexer ist als die Methode `putOrder()`. Das hängt damit zusammen, dass die Zuordnungsannotationen von Spring MVC, inklusive `@PatchMapping` und `@PutMapping`, nur angeben, welche Arten von Anfragen eine Methode behandeln soll. Diese Annotationen schreiben nicht

vor, wie die Anfrage behandelt wird. Selbst wenn `PATCH` semantisch ein partielles Update impliziert, müssen Sie dafür sorgen, dass der Code in der Behandlungsroutine tatsächlich auch eine solche Aktualisierung durchführt.

Im Fall der Methode `putOrder()` haben Sie die vollständigen Daten für eine Bestellung übernommen und gespeichert, wobei Sie sich an die Semantik von HTTP `PUT` gehalten haben. Doch um bei `patchMapping()` der Semantik von HTTP `PATCH` zu entsprechen, ist für den Rumpf der Methode mehr Intelligenz erforderlich. Anstatt die Bestellung vollständig durch die neu gesendeten Daten zu ersetzen, inspiziert die Methode jedes Feld des eintreffenden `Order`-Objekts und wendet alle Werte ungleich `null` auf die vorhandene Bestellung an. Durch dieses Konzept genügt es, wenn der Client nur die Eigenschaften sendet, die geändert werden sollen, und der Server kann vorhandene Daten für alle Eigenschaften, die der Client nicht angegeben hat, beibehalten.

Es gibt mehr als einen Weg, um zu PATCHen

Das in der Methode `patchOrder()` angewandte Patching-Konzept weist eine Reihe von Einschränkungen auf:

- Wenn `null`-Werte dafür stehen sollen, nichts zu ändern, wie kann dann der Client ein Feld kennzeichnen, das auf `null` gesetzt werden soll?
- Es gibt keine Möglichkeit, eine Teilmenge der Elemente einer Auflistung zu entfernen oder hinzuzufügen. Wenn der Client einen Eintrag zu einer Auflistung hinzufügen oder daraus entfernen möchte, muss er die vollständige geänderte Auflistung übermitteln.

Es gibt wirklich keine feste Regel, wie `PATCH`-Anfragen behandelt werden oder wie die eingehenden Daten aussehen sollten. Anstatt die eigentlichen Domänendaten zu senden, könnte ein Client eine Patch-spezifische Beschreibung der anzuwendenden Änderungen übermitteln. Natürlich müsste die Behandlungsroutine für Anfragen so geschrieben werden, dass sie Patch-Anweisungen anstelle der Domänendaten verarbeiten kann.

Beachten Sie sowohl bei `@PutMapping` als auch bei `@PatchMapping`, dass der Anfragepfad auf die zu ändernde Ressource verweist. Auf die gleiche Weise werden Pfade von Methoden behandelt, die mit `@GetMapping` annotiert sind.

Sie wissen nun, wie Sie mit `@GetMapping` und `@Post-Mapping` Ressourcen abrufen und posten. Und Sie haben zwei verschiedene Möglichkeiten kennengelernt, eine Ressource mit `@PutMapping` und `@PatchMapping` zu aktualisieren. Jetzt müssen Sie nur noch Anfragen verarbeiten, um eine Ressource zu löschen.

6.1.4 Daten vom Server löschen

Manchmal werden Daten einfach nicht mehr benötigt. In solchen Fällen sollte ein Client mit einer HTTP-DELETE-Anfrage das Löschen einer Ressource anfordern können.

Die Annotation `@DeleteMapping` von Spring MVC bietet sich an, um Methoden zu deklarieren, die DELETE-Anfragen verarbeiten. Nehmen wir zum Beispiel an, Ihre API soll es ermöglichen, eine Bestellressource zu löschen. Mit der folgenden Controllermethode sollte das gelingen:

```
@DeleteMapping("/{orderId}")
@ResponseStatus(code=HttpStatus.NO_CONTENT)
public void deleteOrder(@PathVariable("orderId") Long orderId) {
  try {
    repo.deleteById(orderId);
  } catch (EmptyResultDataAccessException e) {}
}
```

Mittlerweise sollte Ihnen das Konzept einer weiteren Zuordnungsannotation vertraut sein. Die Annotationen `@GetMapping`, `@PostMapping`, `@PutMapping` und `@PatchMapping` kennen Sie bereits – jede spezifiziert, dass eine Methode die Anfragen für die entsprechenden HTTP-Methoden verarbeiten soll. Es dürfte also nicht überraschen, dass Sie mit `@DeleteMapping` angeben, dass die Methode `deleteOrder()` die DELETE-Anfragen für */orders/{orderId}* verarbeiten soll.

Der Code in der Methode realisiert das eigentliche Löschen einer Bestellung. Hier übernimmt die Methode die ID der Bestellung, die als Pfadvariable in der URL bereitgestellt wird, und übergibt sie an die Methode `deleteById()` des Repositories. Existiert die Bestellung beim Aufruf der Methode, wird sie gelöscht. Andernfalls wird eine `EmptyResultDataAccessException` ausgelöst.

Zwar fange ich die `EmptyResultDataAccessException`-Ausnahme ab, behandle sie aber nicht. Ich gehe davon aus, dass das Ergebnis beim versuchten Löschen einer nicht vorhandenen Ressource das Gleiche ist, als hätte sie vor dem Löschen existiert. Das heißt, die Ressource gibt es schlichtweg nicht. Ob sie vorher existiert hat oder nicht, ist irrelevant. Alternativ hätte ich die Methode `deleteOrder()` so schreiben können, dass sie ein `ResponseEntity`-Objekt mit einem auf `null` gesetzten Körper und dem HTTP-Statuscode NOT FOUND zurückgibt.

Zur Methode `deleteOrder()` ist lediglich noch anzumerken, dass ihre Annotation `@ResponseStatus` den HTTP-Statuscode in der Antwort mit 204 (NO CONTENT) sicherstellt. Es ist nicht notwendig, irgendwelche Ressourcendaten für eine nicht mehr existierende Ressource an den Client zurückzugeben. Antworten auf DELETE-Anfragen besitzen also typischerweise keinen Körper und sollten deshalb einen HTTP-Statuscode übermitteln, damit der Client weiß, dass er keinen Inhalt mehr erwarten kann.

Langsam nimmt Ihre Taco-Cloud-API Gestalt an. Der clientseitige Code kann diese API nun problemlos konsumieren, um Zutaten zu präsentieren, Bestellungen entgegenzunehmen und die neuesten Taco-Kreationen anzuzeigen. Doch es gibt etwas, was Sie tun können, damit der Client Ihre API noch einfacher nutzen kann. Sehen Sie sich an, wie Sie die Taco-Cloud-API für Hypermedia fitmachen können.

6.2 Hypermedia aktivieren

Ihre API ist in ihrer jetzigen Form zwar ziemlich einfach, funktioniert aber, solange der Client, der sie nutzt, das URL-Schema der API beherrscht. Zum Beispiel könnte ein Client fest programmiert sein, um zu wissen, dass er mit einer `GET`-Anfrage für */design/recent* eine Liste der kürzlich kreierten Tacos abrufen kann. Ebenso kann er fest programmiert sein, um zu wissen, dass er die ID eines Tacos in dieser Liste an */design* anhängen kann, um die URL für die betreffende Taco-Ressource zu erhalten.

In API-Clientcode ist es durchaus üblich, fest programmiert URL-Muster und String-Manipulationen zu verwenden. Doch stellen Sie sich einmal vor, was passiert, wenn sich das URL-Schema der API ändert. Der fest programmierte Clientcode hätte veraltetes Wissen über die API und würde deshalb scheitern. Fest codierte API-URLs und die dafür verwendeten String-Manipulationen machen den Clientcode zerbrechlich.

HATEOAS (*Hypermedia as the Engine of Application State*) ist ein Instrument, um selbstbeschreibende APIs zu erstellen, bei denen die von einer API zurückgegebenen Ressourcen Links zu verwandten Ressourcen enthalten. Dadurch finden sich Clients in einer API zurecht, ohne die konkrete URL-Struktur der API zu kennen. Stattdessen verstehen sie *Beziehungen* zwischen den Ressourcen, die von der API bedient werden, und verwenden ihre Kenntnisse von diesen Beziehungen, um die URLs der API zu ermitteln, wenn sie diese Beziehungen durchlaufen.

Nehmen wir zum Beispiel an, ein Client würde eine Liste der neuesten Taco-Kreationen anfordern. Die Liste würde in ihrer rohen Form, ohne Hyperlinks, im Client empfangen werden mit JSON, das wie folgt aussieht (wobei der Kürze wegen alles bis auf den ersten Taco in der Liste abgeschnitten ist):

```
 [
  {
    "id": 4,
    "name": "Veg-Out",
    "createdAt": "2018-01-31T20:15:53.219+0000",
    "ingredients": [
      {"id": "FLTO", "name": "Flour Tortilla", "type": "WRAP"},
      {"id": "COTO", "name": "Corn Tortilla", "type": "WRAP"},
      {"id": "TMTO", "name": "Diced Tomatoes", "type": "VEGGIES"},
      {"id": "LETC", "name": "Lettuce", "type": "VEGGIES"},
      {"id": "SLSA", "name": "Salsa", "type": "SAUCE"}
    ]
  },
  ...
]
```

Möchte der Client eine andere HTTP-Operation auf dem Taco selbst abrufen oder durchführen, müsste er wissen (über feste Programmierung), dass er den Wert der `id`-Eigenschaft an eine URL anfügen kann, deren Pfad */design* lautet. Und wenn er eine HTTP-Operation auf einer der Zutaten durchführen möchte, müsste er wissen, dass er den Wert der `id`-Eigenschaft der Zutat an eine URL mit dem Pfad */ingredients* anfügen könnte. In beiden Fällen müsste er zudem diesem Pfad das Präfix http:// oder https:// und den Hostnamen der API voranstellen.

Wenn dagegen die API Hypermedia-fähig ist, beschreibt die API ihre eigenen URLs, sodass dieses Wissen im Client nicht mehr fest programmiert sein muss. Wenn Hyperlinks eingebettet werden, könnte die gleiche Liste der zuletzt kreierten Tacos wie in Listing 6.3 aussehen.

Listing 6.3 Eine Liste von Taco-Ressourcen, die Hyperlinks enthält

```
{
  "_embedded": {
    "tacoResourceList": [
      {
        "name": "Veg-Out",
        "createdAt": "2018-01-31T20:15:53.219+0000",
        "ingredients": [
          {
            "name": "Flour Tortilla", "type": "WRAP",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/FLTO" }
            }
          },
          {
            "name": "Corn Tortilla", "type": "WRAP",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/COTO" }
            }
          },
          {
            "name": "Diced Tomatoes", "type": "VEGGIES",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/TMTO" }
            }
          },
          {
            "name": "Lettuce", "type": "VEGGIES",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/LETC" }
            }
          },
          {
            "name": "Salsa", "type": "SAUCE",
            "_links": {
              "self": { "href": "http://localhost:8080/ingredients/SLSA" }
            }
          }
        ],
        "_links": {
          "self": { "href": "http://localhost:8080/design/4" }
        }
      },

      ...
    ]
  },
  "_links": {
    "recents": {
```

```
      "href": "http://localhost:8080/design/recent"
    }
  }
}
```

Dieses besondere Format von HATEOAS heißt HAL (*Hypertext Application Language*; http://stateless.co/hal_specification.html). Es ist ein einfaches und häufig verwendetes Format für das Einbetten von Hyperlinks in JSON-Antworten.

Obwohl diese Liste nicht mehr so prägnant ist wie bisher, bietet sie einige nützliche Informationen. Jedes Element in dieser neuen Liste von Tacos schließt eine Eigenschaft `_links` ein, die Hyperlinks enthält, damit der Client durch die API navigieren kann. In diesem Beispiel verfügen sowohl Tacos als auch Zutaten über `self`-Links, um auf diese Ressourcen zu verweisen, und die gesamte Liste besitzt einen Link `recents`, der auf sie selbst verweist.

Sollte eine Clientanwendung eine HTTP-Anfrage gegen einen Taco in der Liste ausführen müssen, lässt sie sich entwickeln ohne Kenntnisse darüber, wie die URL der Taco-Ressource aussieht. Stattdessen ist ihr bekannt, dass sie den `self`-Link abfragen muss, der auf http://localhost:8080/design/4 abgebildet wird. Wenn der Client auf eine bestimmte Zutat zugreifen möchte, muss er nur dem `self`-Link für diese Zutat folgen.

Das Spring-HATEOAS-Projekt realisiert Hyperlink-Unterstützung in Spring. Es bietet eine Reihe von Klassen und Ressourcenassemblern, mit denen Ressourcen mit Links versehen können, bevor Sie sie aus einem Spring-MVC-Controller zurückgeben.

Um Hypermedia in der Taco-Cloud-API zu aktivieren, müssen Sie die Spring-HATEOAS-Starterabhängigkeit zum Build hinzufügen:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
```

Dieser Starter bindet nicht nur Spring HATEOAS in den Klassenpfad des Projekts ein, sondern sorgt auch für die Autokonfiguration, um Spring HATEOAS zu aktivieren. Sie müssen lediglich Ihre Controller überarbeiten, um Ressourcentypen anstelle von Domänentypen zurückzugeben.

Zunächst fügen Sie Hypermedia-Links in die Liste der neuesten Tacos ein, die von einer `GET`-Anfrage an */design/recent* zurückgegeben wird.

6.2.1 Hyperlinks hinzufügen

Spring HATEOAS kennt zwei Haupttypen, die per Hyperlink verknüpfte Ressourcen darstellen: `Resource` und `Resources`. Der Typ `Resource` stellt eine einzelne Ressource dar, der Typ `Resources` ist eine Auflistung von Ressourcen. Beide Typen können Links zu anderen Ressourcen transportieren. Bei der Rückgabe aus einer Spring-MVC-REST-Controllermethode werden die mitgeführten Links in das JSON (oder XML) eingebunden, das der Client empfängt.

Um Hyperlinks zur Liste der neuesten Taco-Kreationen hinzuzufügen, müssen Sie die Methode `recentTacos()` aus Listing 6.2 überarbeiten. Die ursprüngliche Implementierung hat eine `List<Taco>` zurückgegeben, was zu diesem Zeitpunkt in Ordnung war. Jetzt aber brauchen Sie sie, um stattdessen ein `Resources`-Objekt zurückzugeben. Listing 6.4 zeigt eine neue Implementierung von `recentTacos()`, die die ersten Schritte beinhaltet, um Hyperlinks in der Liste der neuesten Tacos zu aktivieren.

Listing 6.4 Hyperlinks zu Ressourcen hinzufügen

```
@GetMapping("/recent")
public Resources<Resource<Taco>> recentTacos() {
  PageRequest page = PageRequest.of(
          0, 12, Sort.by("createdAt").descending());

  List<Taco> tacos = tacoRepo.findAll(page).getContent();
  Resources<Resource<Taco>> recentResources = Resources.wrap(tacos);

  recentResources.add(
      new Link("http://localhost:8080/design/recent", "recents"));
  return recentResources;
}
```

In dieser neuen Version von `recentTacos()` geben Sie die Taco-Liste nicht mehr direkt zurück. Stattdessen verpacken Sie die Taco-Liste mit `Resources.wrap()` als Instanz von `Resources<Resource<Taco>>`, die die Methode letztendlich zurückgibt. Doch bevor Sie das `Resources`-Objekt zurückgeben, fügen Sie einen Link mit dem Beziehungsnamen `recents` und der URL http://localhost:8080/design/recent hinzu. Infolgedessen erscheint das folgende JSON-Fragment in der Ressource, die von der API-Anfrage zurückgegeben wird:

```
"_links": {
  "recents": {
    "href": "http://localhost:8080/design/recent"
  }
}
```

Das ist ein guter Ausgangspunkt, doch es gibt noch einiges zu tun. Zurzeit haben Sie nur einen einzigen Link hinzugefügt, und zwar auf die gesamte Liste; es gibt noch keine Links zu den Taco-Ressourcen selbst oder zu den Zutaten für jeden Taco. Diese Links fügen Sie in Kürze hinzu. Doch zuerst kümmern wir uns um die fest codierte URL, die für den Link `recents` angegeben wurde.

Es ist wirklich nicht sinnvoll, eine derartige URL fest zu codieren. Sofern Sie Ihre Taco-Cloud-Ambitionen nicht darauf beschränkten wollen, die Anwendung nur auf Ihren eigenen Entwicklungscomputern auszuführen, müssen Sie in der Lage sein, URLs zu programmieren, in denen `localhost:8080` nicht von vornherein fix ist. Hier greift Ihnen HATEOAS in Form von Link-Buildern unter die Arme.

Der wohl nützlichste Link-Builder von Spring HATEOAS ist `ControllerLinkBuilder`. Er ist intelligent genug, um den Hostnamen zu ermitteln, ohne dass Sie ihn fest codieren.

Und er stellt eine praktische Fluent API[2] bereit, die Sie dabei unterstützt, Links relativ zur Basis-URL eines Controllers zu erstellen.

Mit `ControllerLinkBuilder` können Sie die fest programmierte `Link`-Erzeugung in `recentTacos()` mit den folgenden Zeilen neu schreiben:

```
Resources<Resource<Taco>> recentResources = Resources.wrap(tacos);
recentResources.add(
  ControllerLinkBuilder.linkTo(DesignTacoController.class)
                       .slash("recent")
                       .withRel("recents"));
```

Jetzt müssen Sie weder den Hostnamen fest codieren noch den Pfad */design* angeben. Stattdessen rufen Sie einen Link auf `DesignTacoController` ab, dessen Basispfad */design* lautet. `ControllerLinkBuilder` verwendet den Basispfad des Controllers als Grundlage für das zu erstellende `Link`-Objekt.

Als Nächstes kommt ein Aufruf einer meiner Lieblingsmethoden in jedem Spring-Projekt: `slash()`. Ich liebe diese Methode, weil sie so prägnant beschreibt, was sie genau tut. Sie fügt buchstäblich einen Schrägstrich (engl. slash, /) und den übergebenen Wert an die URL an. Im Ergebnis wird der Pfad der URL zu */design/recent*.

Schließlich geben Sie einen Beziehungsnamen für den `Link` an. In diesem Beispiel wird die Beziehung `recents` genannt.

Obwohl ich ein großer Fan der Methode `slash()` bin, bietet `ControllerLinkBuilder` noch eine andere Methode, mit der sich fest codierte Elemente bei Link-URLs eliminieren lassen. Statt `slash()` können Sie `linkTo()` aufrufen. Im Aufruf übergeben Sie eine Methode auf dem Controller, um `ControllerLinkBuilder` die Basis-URL sowohl vom Basispfad des Controllers als auch vom zugeordneten Pfad der Methode ableiten zu lassen. Der folgende Code verwendet die Methode `linkTo()` auf diese Weise:

```
Resources<Resource<Taco>> recentResources = Resources.wrap(tacos);
recentResources.add(
        linkTo(methodOn(DesignTacoController.class).recentTacos())
        .withRel("recents"));
```

Hier habe ich absichtlich die Methoden `linkTo()` und `methodOn()` (beide von `ControllerLinkBuilder`) statisch eingebunden, um den Code verständlicher zu halten. Die Methode `methodOn()` übernimmt die Controller-Klasse und erlaubt es Ihnen, die Methode `recentTacos()` aufzurufen, die von `ControllerLinkBuilder` abgefangen und dazu verwendet wird, nicht nur den Basispfad des Controllers zu ermitteln, sondern auch den Pfad, der `recentTacos()` zugeordnet wird. Jetzt wird die gesamte URL aus den Zuordnungen des Controllers abgeleitet und absolut kein Teil davon ist festcodiert. Super!

[2] Fluent Interface – lt. Wikipedia etwa »sprechende Schnittstelle«, d.h. ein Konzept für Programmierschnittstellen, bei dem eine Programmierung fast in Form von natürlicher Sprache möglich ist (siehe https://de.wikipedia.org/wiki/Fluent_Interface).

6.2.2 Ressourcenassembler erstellen

Jetzt müssen Sie Links zu der Taco-Ressource hinzufügen, die in der Liste enthalten ist. Eine Möglichkeit ist es, die einzelnen `Resource<Taco>`-Elemente, die im `Resources`-Objekt transportiert werden, zu durchlaufen und jedem individuell einen `Link` hinzuzufügen. Das ist jedoch etwas mühsam und Sie müssten diesen Schleifencode in der API überall dort wiederholen, wo Sie eine Liste von Taco-Ressourcen zurückgeben.

Wir brauchen eine andere Taktik.

Anstatt die Methode `Resources.wrap()` ein `Resource`-Objekt für jeden Taco in der Liste erzeugen zu lassen, definieren Sie eine Hilfsklasse, die `Taco`-Objekte in ein neues `TacoResource`-Objekt konvertiert. Das `TacoResource`-Objekt sieht fast wie ein `Taco`-Objekt aus, kann aber auch Links transportieren. Listing 6.5 zeigt ein Beispiel für ein `TacoResource`-Objekt.

Listing 6.5 Eine Taco-Ressource, die Domänendaten und eine Liste von Hyperlinks transportiert

```
package tacos.web.api;
import java.util.Date;
import java.util.List;
import org.springframework.hateoas.ResourceSupport;
import lombok.Getter;
import tacos.Ingredient;
import tacos.Taco;

public class TacoResource extends ResourceSupport {

  @Getter
  private final String name;

  @Getter
  private final Date createdAt;

  @Getter
  private final List<Ingredient> ingredients;

  public TacoResource(Taco taco) {
    this.name = taco.getName();
    this.createdAt = taco.getCreatedAt();
    this.ingredients = taco.getIngredients();
  }

}
```

In vielerlei Hinsicht unterscheidet sich `TacoResource` nur wenig vom `Taco`-Domänentyp. Beide verfügen über die Eigenschaften `name`, `createdAt` und `ingredients`. Doch `TacoResource` erweitert `ResourceSupport`, um eine Liste von `Link`-Objekten und Methoden zu erben, mit denen sich die Liste der Links verwalten lässt.

Darüber hinaus beinhaltet `TacoResource` nicht die `id`-Eigenschaft von `Taco`. Das hängt damit zusammen, dass es nicht erforderlich ist, irgendwelche datenbankspezifischen IDs in der

API zugänglich zu machen. Der `self`-Link der Ressource dient als Kennzeichnung für die Ressource aus Sicht eines API-Clients.

Hinweis

Domänen und Ressourcen: getrennt oder gleich? Einige Spring-Entwickler fassen ihre Domänen- und Ressourcentypen möglicherweise in einem einzigen Typ zusammen, indem sie ihre Domänentypen `ResourceSupport` erweitern lassen. Es gibt keine richtige oder falsche Antwort auf die Frage, ob das korrekt ist. Ich ziehe einen separaten Ressourcentyp vor, sodass `Taco` nicht unnötig mit Ressourcenlinks überfrachtet wird, gerade für Anwendungsfälle, die ohne Links auskommen. Außerdem war ich mit einem separaten Ressourcentyp in der Lage, die `id`-Eigenschaft problemlos wegzulassen, sodass sie von der API nicht zugänglich gemacht wird.

`TacoResource` besitzt einen einzigen Konstruktor, der ein `Taco`-Objekt übernimmt und die relevanten Eigenschaften aus dem `Taco` in die eigenen Eigenschaften kopiert. Dadurch lässt sich ein einzelnes `Taco`-Objekt ganz einfach in ein `TacoResource`-Objekt konvertieren. Doch wenn Sie es jetzt dabei bewenden lassen, müssten Sie immer noch eine Liste von `Taco`-Objekten in einer Schleife zu `Resources<TacoResource>` konvertieren.

Um das Konvertieren von `Taco`-Objekten in `TacoResource`-Objekte zu unterstützen, erstellen Sie auch einen Ressourcenassembler. Listing 6.6 zeigt, was Sie benötigen.

Listing 6.6 Ein Ressourcenassembler, der Taco-Ressourcen zusammenstellt

```
package tacos.web.api;

import org.springframework.hateoas.mvc.ResourceAssemblerSupport;

import tacos.Taco;

public class TacoResourceAssembler
       extends ResourceAssemblerSupport<Taco, TacoResource> {

  public TacoResourceAssembler() {
    super(DesignTacoController.class, TacoResource.class);
  }

  @Override
  protected TacoResource instantiateResource(Taco taco) {
    return new TacoResource(taco);
  }
  @Override
  public TacoResource toResource(Taco taco) {
    return createResourceWithId(taco.getId(), taco);
  }
}
```

Der Standardkonstruktor von `TacoResourceAssembler` informiert die Superklasse (`ResourceAssemblerSupport`) darüber, dass er mit `DesignTacoController` den Basispfad für alle URLs in den Links ermittelt, die er beim Anlegen einer `TacoResource` erstellt.

Die Methode `instantiateResource()` wird überschrieben, um ein `TacoResource`-Objekt für einen übergebenen `Taco` zu instanziieren. Diese Methode wäre optional, wenn `TacoResource` über einen Standardkonstruktor verfügen würde. Hier jedoch verlangt `TacoResource` eine Konstruktion mit einem `Taco`, sodass Sie die Methode überschreiben müssen.

Schließlich ist die Methode `toResource()` die einzige Methode, die unbedingt erforderlich ist, wenn `ResourceAssemblerSupport` erweitert wird. Hier weisen Sie die Methode an, ein `TacoResource`-Objekt aus einem `Taco` zu erzeugen und es automatisch mit einem `self`-Link zu versehen, wobei die URL aus der `id`-Eigenschaft des `Taco`-Objekts abgeleitet wird.

Nach außen hin scheint `toResource()` einen ähnlichen Zweck wie `instantiateResource()` zu haben, doch die Aufgaben der Methoden unterscheiden sich etwas. Während `instantiateResource()` nur dafür gedacht ist, ein `Resource`-Objekt zu instanziieren, soll `toResource()` nicht nur das `Resource`-Objekt erzeugen, sondern es auch mit Links füllen. Hinter den Kulissen ruft `toResource()` die Methode `instantiateResource()` auf.

Passen Sie nun die Methode `recentTacos()` an, um den `TacoResourceAssembler` zu nutzen:

```
@GetMapping("/recent")
public Resources<TacoResource> recentTacos() {
  PageRequest page = PageRequest.of(
          0, 12, Sort.by("createdAt").descending());
  List<Taco> tacos = tacoRepo.findAll(page).getContent();
  List<TacoResource> tacoResources =
      new TacoResourceAssembler().toResources(tacos);
  Resources<TacoResource> recentResources =
      new Resources<TacoResource>(tacoResources);
  recentResources.add(
      linkTo(methodOn(DesignTacoController.class).recentTacos())
      .withRel("recents"));
  return recentResources;
}
```

Anstatt ein `Resources<Resource<Taco>>` zurückzugeben, gibt `recentTacos()` jetzt ein `Resources<TacoResource>` zurück, um von Ihrem neuen Typ `TacoResource` zu profitieren. Nachdem Sie die Tacos aus dem Repository abgerufen haben, übergeben Sie die Liste der `Taco`-Objekte an die `toResources()`-Methode auf einem `TacoResourceAssembler`. Diese praktische Methode durchläuft alle `Taco`-Objekte und ruft dabei die Methode `toResource()` auf, die Sie in `TacoResourceAssembler` überschrieben haben, um eine Liste von `TacoResource`-Objekten zu erstellen.

Mit dieser `TacoResource`-Liste erstellen Sie als Nächstes ein `Resources<TacoResource>`-Objekt und füllen es dann mit den `recents`-Links wie in der vorherigen Version von `recentTacos()`.

An dieser Stelle erzeugt eine `GET`-Anfrage an */design/recent* eine Liste von Tacos, die jeweils einen `self`-Link und einen `recents`-Link auf der Liste selbst enthalten. Die Zutaten werden aber immer noch ohne Link sein. Um dies zu ändern, erstellen Sie einen neuen Ressourcenassembler für Zutaten:

```
package tacos.web.api;
import org.springframework.hateoas.mvc.ResourceAssemblerSupport;
import tacos.Ingredient;

class IngredientResourceAssembler extends
          ResourceAssemblerSupport<Ingredient, IngredientResource> {

  public IngredientResourceAssembler() {
    super(IngredientController2.class, IngredientResource.class);
  }

  @Override
  public IngredientResource toResource(Ingredient ingredient) {
    return createResourceWithId(ingredient.getId(), ingredient);
  }

  @Override
  protected IngredientResource instantiateResource(
                                          Ingredient ingredient) {
    return new IngredientResource(ingredient);
  }

}
```

Wie aus dem Code hervorgeht, entspricht `IngredientResourceAssembler` weitgehend dem `TacoResourceAssembler`, arbeitet aber mit `Ingredient`- und `IngredientResource`-Objekten anstelle von `Taco`- und `TacoResource`-Objekten. Und da wir gerade von `IngredientResource` sprechen, diese Klasse sieht so aus:

```
package tacos.web.api;
import org.springframework.hateoas.ResourceSupport;
import lombok.Getter;
import tacos.Ingredient;
import tacos.Ingredient.Type;

public class IngredientResource extends ResourceSupport {

  @Getter
  private String name;

  @Getter
  private Type type;

  public IngredientResource(Ingredient ingredient) {
    this.name = ingredient.getName();
    this.type = ingredient.getType();
  }

}
```

Wie bei `TacoResource` erweitert `IngredientResource` die Klasse `ResourceSupport` und kopiert relevante Eigenschaften vom Domänentyp in seinen eigenen Satz von Eigenschaften (mit Ausnahme der Eigenschaft `id`).

Nun müssen wir noch die Klasse `TacoResource` etwas verändern, damit sie `IngredientResource`-Objekte statt `Ingredient`-Objekte übernimmt:

```
package tacos.web.api;
import java.util.Date;
import java.util.List;
import org.springframework.hateoas.ResourceSupport;
import lombok.Getter;
import tacos.Taco;

public class TacoResource extends ResourceSupport {

  private static final IngredientResourceAssembler
           ingredientAssembler = new IngredientResourceAssembler();

  @Getter
  private final String name;

  @Getter
  private final Date createdAt;

  @Getter
  private final List<IngredientResource> ingredients;

  public TacoResource(Taco taco) {
    this.name = taco.getName();
    this.createdAt = taco.getCreatedAt();
    this.ingredients =
        ingredientAssembler.toResources(taco.getIngredients());
  }

}
```

Diese neue Version von `TacoResource` erzeugt eine statische finale Instanz von `IngredientResourceAssembler` und verwendet dessen Methode `toResource()`, um die Liste von `Ingredient`-Objekten eines bestimmten `Taco`-Objekts in eine Liste von `IngredientResource`-Objekten zu konvertieren.

Nunmehr ist Ihre neue Taco-Liste vollständig mit Hyperlinks ausgestattet, und nicht nur für sich selbst (über den Link `recents`), sondern auch für alle ihre Taco-Einträge und die Zutaten dieser Tacos. Die Antwort sollte wie das JSON in Listing 6.3 aussehen.

Prinzipiell könnten Sie hier Schluss machen und zum nächsten Thema übergehen. Doch ich möchte noch etwas behandeln, was mich bei Listing 6.3 stört.

6.2.3 Eingebettete Beziehungen benennen

In Listing 6.3 sehen die Elemente der obersten Ebene wie folgt aus:

```
{
  "_embedded": {
```

```
    "tacoResourceList": [
      ...
    ]
  }
}
```

Insbesondere möchte ich Sie auf den Namen `tacoResourceList` unter `embedded` aufmerksam machen. Der Name leitet sich von der Tatsache ab, dass das `Resources`-Objekt aus einer `List<TacoResource>` erzeugt wurde. Auch wenn es nicht wahrscheinlich ist, doch wenn Sie den Namen der Klasse `TacoResource` in etwas anderes refaktorisieren möchten, würde sich der Feldname im resultierenden JSON entsprechend ändern. Dadurch würden höchstwahrscheinlich alle Clients versagen, deren Code sich auf diesen Namen verlässt.

Die Annotation `@Relation` kann dabei helfen, die JSON-Feldnamen von den in Java definierten Klassennamen des Ressourcentyps zu entkoppeln. Indem Sie `TacoResource` mit `@Relation` annotieren, können Sie festlegen, wie Spring HATEOAS das Feld im resultierenden JSON benennen soll:

```
@Relation(value="taco", collectionRelation="tacos")
public class TacoResource extends ResourceSupport {
  ...
}
```

Hiermit legen Sie fest, dass eine Liste von `TacoResource`-Objekten mit `tacos` zu benennen ist, wenn sie in einem `Resources`-Objekt verwendet wird. Und obwohl Sie in unserer API keinen Gebrauch davon machen, sollte ein einzelnes `TacoResource`-Objekt in JSON als `taco` referenziert werden.

Infolgedessen sieht das von */design/recent* zurückgegebene JSON nun folgendermaßen aus (unabhängig davon, welches Refactoring Sie auf `TacoResource` durchführen oder nicht):

```
{
  "_embedded": {
    "tacos": [
      ...
    ]
  }
}
```

Mit Spring HATEOAS können Sie ziemlich einfach Links auf Ihre API hinzuzufügen. Dennoch habe ich einige Codezeilen ergänzt, die Sie anderweitig nicht benötigen. Aus diesem Grund entscheiden sich einige Entwickler vielleicht, sich in ihren APIs nicht mit HATEOAS herumzuschlagen, selbst wenn das heißt, dass der Clientcode versagt, wenn sich das URL-Schema der API ändert. Ich empfehle Ihnen, HATEOAS ernst zu nehmen und nicht auf den faulen Ausweg zu setzen, indem Sie in Ihre Ressourcen keine Hyperlinks einbinden.

Möchten Sie aber partout faul sein, gibt es vielleicht ein Win-Win-Szenario für Sie, wenn Sie Spring Data für Ihre Repositories verwenden. Sehen wir uns an, wie Spring Data REST Ihnen helfen kann, APIs automatisch zu erzeugen, und zwar basierend auf den Datenspeichern, die Sie mit Spring Data in Kapitel 3 angelegt haben.

6.3 Datengestützte Dienste aktivieren

Wie Kapitel 3 gezeigt hat, erzeugt Spring Data Repository-Implementierungen, basierend auf den Schnittstellen, die Sie in Ihrem Code definieren. Doch Spring Data hat noch einen anderen Trick auf Lager, der Sie dabei unterstützen kann, APIs für Ihre Anwendung zu definieren.

Als weiteres Mitglied der Spring-Data-Familie erzeugt Spring Data REST automatisch REST APIs für Repositories, die von Spring Data erstellt wurden. Es ist nicht viel mehr erforderlich, als Spring Data REST zu Ihrem Build hinzuzufügen, und schon erhalten Sie eine API mit Operationen für sämtliche Repository-Schnittstellen, die Sie definiert haben.

Um Spring Data REST zu verwenden, fügen Sie die folgende Abhängigkeit in Ihren Build ein:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

Ob Sie es glauben oder nicht, mehr ist nicht erforderlich, um eine REST API in einem Projekt zugänglich zu machen, das bereits Spring Data für automatische Repositories verwendet. Einfach den Spring Data REST-Starter in den Build einbinden und schon erhält die Anwendung Autokonfiguration, die automatisches Erstellen einer REST API für jedes Repository ermöglicht, das durch Spring Data erzeugt wurde (inklusive Spring Data JPA, Spring Data Mongo usw.).

Die REST-Endpunkte, die Spring Data REST erzeugt, sind mindestens so gut wie (und möglicherweise sogar besser als) die, die Sie selbst erstellt haben. Und bevor Sie weitermachen: An dieser Stelle können Sie einige Abbrucharbeiten vornehmen und alle Klassen mit der Annotation `@RestController` entfernen, die Sie bis jetzt erstellt haben.

Um die Endpunkte auszuprobieren, die Spring Data REST bereitgestellt hat, starten Sie die Anwendung und stöbern in einigen URLs. Aufbauend auf den Repositories, die Sie bereits für Taco Cloud definiert haben, sollten Sie nun `GET`-Anfragen nach Tacos, Zutaten, Bestellungen und Benutzern durchführen können.

So können Sie beispielsweise eine Liste aller Zutaten erhalten, indem Sie eine GET-Anfrage an */ingredients* richten. Per `curl` bekommen Sie etwas, das wie folgt aussieht (gekürzt, um nur die erste Zutat anzuzeigen):

```
$ curl localhost:8080/ingredients
{
  "_embedded" : {
    "ingredients" : [ {
      "name" : "Flour Tortilla",
      "type" : "WRAP",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/ingredients/FLTO"
        },
        "ingredient" : {
```

```
          "href" : "http://localhost:8080/ingredients/FLTO"
        }
      }
    },
    ...
    ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/ingredients"
    },
    "profile" : {
      "href" : "http://localhost:8080/profile/ingredients"
    }
  }
}
```

Wow! Indem Sie lediglich eine Abhängigkeit in Ihren Build einfügen, bekommen Sie nicht nur einen Endpunkt für Zutaten, sondern die zurückgegebenen Ressourcen enthalten auch Hyperlinks! Wenn Sie sich als Client dieser API ausgeben, können Sie auch per `curl` dem `self`-Link für die Weizen-Tortilla (Flour Tortilla) folgen:

```
$ curl http://localhost:8080/ingredients/FLTO
{
  "name" : "Flour Tortilla",
  "type" : "WRAP",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/ingredients/FLTO"
    },
    "ingredient" : {
      "href" : "http://localhost:8080/ingredients/FLTO"
    }
  }
}
```

Um nicht zu sehr abgelenkt zu werden, verschwenden wir in diesem Buch keine Zeit mehr damit, jeden einzelnen Endpunkt und jede Option zu untersuchen, die Spring Data REST erzeugt hat. Aber Sie sollten wissen, dass auch die Methoden `POST`, `PUT` und `DELETE` für die erzeugten Endpunkte unterstützt werden. Es stimmt: Sie können mit `POST` an */ingredients* eine neue Zutat erzeugen und mit `DELETE` an */ingredients/FLTO* die Weizen-Tortillas aus dem Menü entfernen.

Es empfiehlt sich außerdem, einen Basispfad für die API einzurichten, damit ihre Endpunkte eindeutig sind und nicht mit anderen Controllern kollidieren, die Sie schreiben. (Wenn Sie den `IngredientsController`, den Sie zuvor erstellt haben, nicht entfernen, kommt es tatsächlich zu einem Konflikt mit dem von Spring Data REST bereitgestellten Endpunkt */ingredients*.) Um den Basispfad für die API anzupassen, setzen Sie die Eigenschaft `spring.data.rest.base-path`:

```
spring:
  data:
    rest:
      base-path: /api
```

Damit wird der Basispfad für die Endpunkte von Spring Data REST auf */api* eingestellt. Folglich lautet der Endpunkt für die Zutaten jetzt */api/ingredients*. Fordern Sie nun eine Liste von Tacos an, um diesen neuen Basispfad auszuprobieren:

```
$ curl http://localhost:8080/api/tacos
{
  "timestamp": "2018-02-11T16:22:12.381+0000",
  "status": 404,
  "error": "Not Found",
  "message": "No message available",
  "path": "/api/tacos"
}
```

Oje! Das hat nicht wie erwartet funktioniert. Sie haben eine `Ingredient`-Entität und eine `IngredientRepository`-Schnittstelle, die Spring Data REST mit einem Endpunkt */api/ingredients* zugänglich macht. Wenn Sie also eine `Taco`-Entität und eine `TacoRepository`-Schnittstelle haben, warum gibt Ihnen Spring Data REST dann keinen Endpunkt */api/tacos*?

6.3.1 Ressourcenpfade und Beziehungsnamen anpassen

Eigentlich *gibt* Spring Data REST Ihnen einen Endpunkt, um mit Tacos zu arbeiten. Doch so clever Spring Data REST auch sein mag, es zeigt sich selbst etwas weniger cool darin, wie es den Tacos-Endpunkt zugänglich macht.

Wenn Spring Data REST Endpunkte für Spring-Data-Repositories erstellt, versucht es, die zugeordnete Entitätsklasse zu pluralisieren. Für die Entität `Ingredient` lautet der Endpunkt */ingredients*. Die Endpunkte für die Entitäten `Order` und `User` heißen */orders* bzw. */users*. So weit, so gut.

Manchmal aber kommt Spring Data REST mit einem Wort nicht klar und bildet keine korrekte Pluralversion. Das ist zum Beispiel bei »taco« der Fall. Spring Data REST pluralisiert das Wort »taco« zu »tacoes«. Um also eine Anfrage nach Tacos zu stellen, müssen Sie dieses Spiel mitmachen und */api/tacoes* anfragen:

```
% curl localhost:8080/api/tacoes
{
  "_embedded" : {
    "tacoes" : [ {
      "name" : "Carnivore",
      "createdAt" : "2018-02-11T17:01:32.999+0000",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/api/tacoes/2"
        },
```

```
        "taco" : {
          "href" : "http://localhost:8080/api/tacoes/2"
        },
        "ingredients" : {
          "href" : "http://localhost:8080/api/tacoes/2/ingredients"
        }
      }
    }]
  },
  "page" : {
    "size" : 20,
    "totalElements" : 3,
    "totalPages" : 1,
    "number" : 0
  }
}
```

Doch woher weiß ich überhaupt, dass »taco« fälschlicherweise zu »tacoes« pluralisiert wird? Es zeigt sich, dass Spring Data REST auch eine Home-Ressource zugänglich macht, die Links für alle offengelegten Endpunkte enthält. Führen Sie einfach eine `GET`-Anfrage zum API-Basispfad aus, um diese Links zu erhalten:

```
$ curl localhost:8080/api
{
  "_links" : {
    "orders" : {
      "href" : "http://localhost:8080/api/orders"
    },
    "ingredients" : {
      "href" : "http://localhost:8080/api/ingredients"
    },
    "tacoes" : {
      "href" : "http://localhost:8080/api/tacoes{?page,size,sort}",
      "templated" : true
    },
    "users" : {
      "href" : "http://localhost:8080/api/users"
    },
    "profile" : {
      "href" : "http://localhost:8080/api/profile"
    }
  }
}
```

Die Home-Ressource zeigt die Links für alle Ihre Entitäten. Alles sieht gut aus, bis auf den `tacoes`-Link, bei dem sowohl der Beziehungsname als auch die URL die eigenartige Pluralisierung von »taco« aufweisen.

Allerdings müssen Sie sich nicht mit dieser kleinen Laune von Spring Data REST abfinden. Mit einer Annotation an der Klasse `Taco` können Sie sowohl den Beziehungsnamen als auch diesen Pfad anpassen:

```
@Data
@Entity
@RestResource(rel="tacos", path="tacos")
public class Taco {
  ...
}
```

Mit der Annotation `@RestResource` können Sie der Entität jeden gewünschten Namen geben. In diesem Fall setzen Sie beide Bezeichner auf »tacos«. Wenn Sie jetzt die Home-Ressource abfragen, sehen Sie, dass der `tacos`-Link korrekt pluralisiert ist:

```
"tacos" : {
  "href" : "http://localhost:8080/api/tacos{?page,size,sort}",
  "templated" : true
},
```

Damit wird auch der Pfad für den Endpunkt richtig eingeordnet, sodass Sie Anfragen gegen */api/tacos* ausführen können, um mit Taco-Ressourcen zu arbeiten.

Apropos ordnen: Sehen Sie sich nun an, wie Sie die Ergebnisse von Spring-Data-REST-Endpunkten sortieren können.

6.3.2 Paging und Sortieren

Sicherlich haben Sie bemerkt, dass alle Links in der Home-Ressource die optionalen Parameter `page`, `size` und `sort` besitzen. Standardmäßig geben Anfragen an eine Auflistungsressource wie zum Beispiel */api/tacos* bis zu 20 Elemente pro Seite von der ersten Seite zurück. Die angezeigte Seite und die Seitengröße können Sie allerdings festlegen, wenn Sie die Parameter `page` und `size` in Ihrer Anfrage angeben.

So können Sie zum Beispiel mit der folgenden `GET`-Anfrage (per `curl`) die erste Seite der Tacos mit einer Seitengröße von 5 abrufen:

```
$ curl "localhost:8080/api/tacos?size=5"
```

Sofern mehr als fünf Tacos gespeichert sind, können Sie die zweite Seite der Tacos abrufen, indem Sie den Parameter `page` hinzufügen:

```
$ curl "localhost:8080/api/tacos?size=5&page=1"
```

Da die Nummerierung der Seiten nullbasiert erfolgt, setzen Sie den Parameter `page` auf 1, um die zweite Seite abzurufen. (Weil viele Befehlszeilen-Shells über das Et-Zeichen in der Anfrage stolpern, habe ich die gesamte URL im obigen `curl`-Befehl in Anführungszeichen gesetzt.)

Zwar könnten Sie diese Parameter per String-Manipulation in die URL einfügen, doch HATEOAS unterstützt Sie hier mit Links für die erste, letzte, nächste und vorhergehende Seite in der Antwort:

```
"_links" : {
  "first" : {
    "href" : "http://localhost:8080/api/tacos?page=0&size=5"
  },
  "self" : {
    "href" : "http://localhost:8080/api/tacos"
  },
  "next" : {
    "href" : "http://localhost:8080/api/tacos?page=1&size=5"
  },
  "last" : {
    "href" : "http://localhost:8080/api/tacos?page=2&size=5"
  },
  "profile" : {
    "href" : "http://localhost:8080/api/profile/tacos"
  },
  "recents" : {
    "href" : "http://localhost:8080/api/tacos/recent"
  }
}
```

Mit diesen Links muss ein Client der API weder verfolgen, auf welcher Seite er sich befindet, noch die Parameter mit der URL verketten. Stattdessen muss er lediglich wissen, wo er diese Links für die Seitennavigation nach ihrem Namen suchen kann, und ihnen folgen.

Der Parameter `sort` erlaubt es Ihnen, die Ergebnisliste nach einer beliebigen Eigenschaft der Entität zu sortieren. Angenommen, Sie möchten die zwölf zuletzt erzeugten Tacos abrufen, um sie in der Benutzeroberfläche anzuzeigen. Mit dem folgenden Mix aus Paging- und Sortierparametern lässt sich dies bewerkstelligen:

```
$ curl "localhost:8080/api/tacos?sort=createdAt,desc&page=0&size=12"
```

Hier spezifiziert der Parameter `sort`, dass nach der Eigenschaft `createdDate` zu sortieren ist, und zwar in absteigender Richtung (sodass die neuesten Tacos zuerst erscheinen). Die Parameter `page` und `size` legen fest, dass Sie die erste Seite mit zwölf Tacos sehen sollten.

Dies ist genau das, was die Benutzeroberfläche benötigt, um die zuletzt kreierten Tacos anzuzeigen. Es ist ungefähr das Gleiche wie der Endpunkt */design/recent*, den Sie in `DesignTacoController` weiter vorn in diesem Kapitel erstellt haben.

Es gibt jedoch ein kleines Problem. Der UI-Code muss fest programmiert werden, um die Liste der Tacos mit diesen Parametern abzurufen. Sicherlich funktioniert das. Doch Sie machen den Client etwas spröde, weil er wissen muss, wie eine API-Anfrage zu konstruieren ist. Es wäre toll, wenn der Client die URL auf einer Liste von Links nachschlagen könnte. Und noch großartiger wäre es, wenn die URL prägnanter wäre, etwa wie der Endpunkt */design/recent*, den Sie weiter vorn erstellt haben.

6.3.3 Benutzerdefinierte Endpunkte hinzufügen

Spring Data REST ist hervorragend geeignet, um Endpunkte zu erstellen, über die sich CRUD-Operationen gegen Spring-Data-Repositories durchführen lassen. Manchmal aber müssen Sie sich von der standardmäßigen CRUD API lösen und einen Endpunkt erzeugen, der direkt zum Kern des Problems führt.

Nichts hält Sie davon ab, einen beliebigen Endpunkt zu erstellen, mit dem Sie in einer Bean, die mit `@RestController` annotiert ist, das ergänzen, was Spring Data REST automatisch erzeugt. Tatsächlich könnten Sie den `DesignTacoController` von weiter vorn in diesem Kapitel wiederbeleben und er würde immer noch neben den von Spring Data REST bereitgestellten Endpunkten funktionieren.

Doch wenn Sie Ihre eigenen API-Controller schreiben, scheinen deren Endpunkte in mehrfacher Hinsicht von den Spring Data REST-Endpunkten ein wenig abgekoppelt zu sein:

- Ihre eigenen Controller-Endpunkte werden nicht unter dem Basispfad von Spring Data REST abgebildet. Zwar könnten Sie erzwingen, dass deren Zuordnungen einen Basispfad – den Spring Data REST-Basispfad – als Präfix bekommen, doch wenn sich der Basispfad ändern sollte, müssten Sie die Zuordnungen des Controllers entsprechend bearbeiten.
- Alle Endpunkte, die Sie in Ihren eigenen Controllern definieren, werden nicht automatisch als Hyperlinks in die von den Spring-Data-REST-Endpunkten zurückgegebenen Ressourcen eingebunden. Demzufolge werden Clients nicht in der Lage sein, Ihre benutzerdefinierten Endpunkte mit einem Beziehungsnamen zu erkennen.

Zunächst kümmern wir uns um das Problem mit dem Basispfad. Spring Data REST führt mit `@RepositoryRestController` eine neue Annotation ein. Damit lassen sich Controller-Klassen annotieren, deren Zuordnungen einen Basispfad annehmen sollten, der demjenigen entspricht, der für Spring-Data-REST-Endpunkte konfiguriert ist. Einfach ausgedrückt wird der Pfad aller Zuordnungen in einem mit `@RepositoryRestController` annotierten Controller mit einem Präfix versehen, dessen Wert sich aus der Eigenschaft `spring.data.rest.base-path` ergibt (die Sie als */api* konfiguriert haben).

Anstatt den `DesignTacoController`, der mehrere Handler-Methoden umfasst, die Sie nicht brauchen, wiederzubeleben, erstellen Sie einen neuen Controller, der nur die Methode `recentTacos()` enthält. Die Klasse `RecentTacosController` in Listing 6.7 ist mit `@RepositoryRestController` annotiert, um den Basispfad von Spring Data REST für seine Anfragezuordnungen zu übernehmen.

Listing 6.7 Den Basispfad von Spring Data REST auf einen Controller anwenden

```
package tacos.web.api;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.*;
import java.util.List;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.data.rest.webmvc.RepositoryRestController;
import org.springframework.hateoas.Resources;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
```

```
import tacos.Taco;
import tacos.data.TacoRepository;

@RepositoryRestController
public class RecentTacosController {

  private TacoRepository tacoRepo;

  public RecentTacosController(TacoRepository tacoRepo) {
    this.tacoRepo = tacoRepo;
  }

  @GetMapping(path="/tacos/recent", produces="application/hal+json")
  public ResponseEntity<Resources<TacoResource>> recentTacos() {
    PageRequest page = PageRequest.of(
                          0, 12, Sort.by("createdAt").descending());
    List<Taco> tacos = tacoRepo.findAll(page).getContent();

    List<TacoResource> tacoResources =
        new TacoResourceAssembler().toResources(tacos);
    Resources<TacoResource> recentResources =
            new Resources<TacoResource>(tacoResources);
    recentResources.add(
        linkTo(methodOn(RecentTacosController.class).recentTacos())
            .withRel("recents"));
    return new ResponseEntity<>(recentResources, HttpStatus.OK);
  }

}
```

Selbst wenn `@GetMapping` dem Pfad */tacos/recent* zugeordnet ist, stellt die Annotation `@RepositoryRestController` auf Klassenebene sicher, dass er als Präfix den Klassenpfad von Spring Data REST erhält. Entsprechend der aktuellen Konfiguration verarbeitet die Methode `recentTacos()` die `GET`-Anfragen für */api/tacos/recent*.

Beachten Sie, dass `@RepositoryRestController` zwar ähnlich wie `@RestController` benannt ist, aber nicht die gleiche Semantik wie `@RestController` beinhaltet. Insbesondere stellt sie nicht sicher, dass die Rückgabewerte von Handler-Methoden automatisch in den Körper der Antwort geschrieben werden. Demzufolge müssen Sie entweder die Methode mit `@ResponseBody` annotieren oder ein `ResponseEntity`-Objekt zurückgeben, das die Antwortdaten einhüllt. Hier wird die Version mit `ResponseEntity` verwendet.

Wenn `RecentTacosController` im Spiel ist, geben Anfragen nach */api/tacos/recent* bis zu 15 der zuletzt kreierten Tacos zurück, ohne dass in der URL Parameter für Paging und Sortieren erforderlich sind. Allerdings erscheint die URL immer noch nicht in der Hyperlink-Liste, wenn Sie */api/tacos* abfragen. Das bringen wir jetzt in Ordnung.

6.3.4 Benutzerdefinierte Hyperlinks zu Spring-Data-Endpunkten hinzufügen

Wenn sich der Endpunkt für die neuesten Tacos nicht unter den Hyperlinks befindet, die von */api/tacos* zurückgegeben werden, woher soll dann ein Client wissen, wie er die neuesten Tacos abrufen kann? Er muss es entweder erraten oder die Paging- und Sortierparameter verwenden. So oder so, er wird im Clientcode fest programmiert, was nicht ideal ist.

Wenn Sie aber eine Ressourcenprozessor-Bean deklarieren, können Sie Links zur Liste der Links hinzufügen, die Spring Data REST automatisch einbindet. Spring Data HATEOAS stellt mit `ResourceProcessor` eine Schnittstelle bereit, um Ressourcen zu manipulieren, bevor sie über die API zurückgegeben werden. Für Ihre Zwecke brauchen Sie eine Implementierung von `ResourceProcessor`, die einen `recents`-Link zu jeder Ressource vom Typ `ResourceProcessor` hinzufügt (dem Typ, der für den Endpunkt */api/tacos* zurückgegeben wird). Listing 6.8 zeigt eine Bean-Deklarationsmethode, die einen derartigen `ResourceProcessor` definiert.

Listing 6.8 Benutzerdefinierte Links zu einem Spring-Data-REST-Endpunkt hinzufügen

```
@Bean
public ResourceProcessor<PagedResources<Resource<Taco>>>
  tacoProcessor(EntityLinks links) {

  return new ResourceProcessor<PagedResources<Resource<Taco>>>() {
    @Override
    public PagedResources<Resource<Taco>> process(
                        PagedResources<Resource<Taco>> resource) {
      resource.add(
          links.linkFor(Taco.class)
               .slash("recent")
               .withRel("recents"));
      return resource;
    }
  };
}
```

Der `ResourceProcessor` in Listing 6.8 ist als anonyme innere Klasse definiert und als Bean deklariert, die im Spring-Anwendungskontext erstellt wird. Spring HATEOAS erkennt diese Bean (wie auch alle anderen Beans vom Typ `ResourceProcessor`) automatisch und wendet sie auf die passenden Ressourcen an. Wenn ein Controller wie in diesem Fall eine `PagedResources<Resource<Taco>>` zurückgibt, erhält sie einen Link für die zuletzt kreierten Tacos. Dies schließt die Antwort auf Anfragen für */api/tacos* ein.

6.4 Zusammenfassung

- REST-Endpunkte lassen sich mit Spring MVC erstellen, mit Controllern, die dem gleichen Programmiermodell wie browserorientierte Controller folgen.
- Handler-Methoden von Controllern können entweder mit `@ResponseBody` annotiert werden oder `ResponseEntity`-Objekte zurückgeben, um das Modell und die View zu umgehen und Daten direkt in den Body der Antwort zu schreiben.
- Die Annotation `@RestController` vereinfacht REST-Controller, weil es nicht mehr erforderlich ist, die Annotation `@ResponseBody` auf Handler-Methoden anzuwenden.
- Spring HATEOAS ermöglicht es, Ressourcen, die von Spring MVC-Controllern zurückgegeben werden, mit Hyperlinks zu versehen.
- Spring-Data-Repositories können automatisch mithilfe von Spring Data REST als REST APIs zugänglich gemacht werden.