

HANSER



Leseprobe

zu

„Spring im Einsatz“

von Craig Walls

Print-ISBN: 978-3-446-45512-2

E-Book-ISBN: 978-3-446-45731-7

E-Pub-ISBN: 978-3-446-46323-3

Weitere Informationen und Bestellungen unter
<http://www.hanser-fachbuch.de/978-3-446-45512-2>
sowie im Buchhandel

© Carl Hanser Verlag, München

Inhalt

Vorwort	XIII
Danksagungen	XV
Über dieses Buch	XVII
1 Erste Schritte mit Spring	3
1.1 Was ist Spring?	4
1.2 Eine Spring-Anwendung initialisieren	6
1.2.1 Ein Spring-Projekt mit der Spring Tool Suite initialisieren	7
1.2.2 Die Spring-Projektstruktur untersuchen	11
1.3 Eine Spring-Anwendung schreiben	17
1.3.1 Web-Requests verarbeiten	17
1.3.2 Die View definieren	19
1.3.3 Den Controller testen	20
1.3.4 Die Anwendung erstellen und ausführen	21
1.3.5 Spring Boot DevTools kennenlernen	23
1.3.6 Rückblick	25
1.4 Die Spring-Landschaft im Überblick	26
1.4.1 Der Core des Spring Frameworks	26
1.4.2 Spring Boot	27
1.4.3 Spring Data	27
1.4.4 Spring Security	28
1.4.5 Spring Integration und Spring Batch	28
1.4.6 Spring Cloud	28
1.5 Zusammenfassung	29
2 Webanwendungen entwickeln	31
2.1 Informationen anzeigen	31
2.1.1 Die Domäne einrichten	33
2.1.2 Eine Controller-Klasse erstellen	34
2.1.3 Die View entwerfen	38
2.2 Formularübermittlungen verarbeiten	43
2.3 Formulareingaben validieren	49
2.3.1 Validierungsregeln deklarieren	49
2.3.2 Validierung bei Formularbindung durchführen	52

2.3.3	Validierungsfehler anzeigen	53
2.4	Mit View-Controllern arbeiten	55
2.5	Eine View-Template-Bibliothek auswählen	57
2.5.1	Vorlagen zwischenspeichern	59
2.6	Zusammenfassung	60
3	Mit Daten arbeiten	61
3.1	Daten mit JDBC lesen und schreiben	61
3.1.1	Die Domäne für Persistenz anpassen	64
3.1.2	Mit JdbcTemplate arbeiten	65
3.1.3	Ein Schema definieren und Daten im Voraus laden	69
3.1.4	Daten einfügen	72
3.2	Daten mit Spring Data JPA persistent speichern	81
3.2.1	Spring Data JPA zum Projekt hinzufügen	81
3.2.2	Die Domäne als Entitäten annotieren	82
3.2.3	JPA-Repositories deklarieren	86
3.2.4	JPA-Repositories anpassen	87
3.3	Zusammenfassung	90
4	Zugriffskontrolle mit Spring Security	91
4.1	Spring Security aktivieren	91
4.2	Spring Security konfigurieren	94
4.2.1	Speicherinterner Benutzerspeicher	96
4.2.2	JDBC-basierter Benutzerspeicher	97
4.2.3	LDAP-gestützter Benutzerspeicher	100
4.2.4	Benutzerauthentifizierung anpassen	104
4.3	Webanfragen sichern	112
4.3.1	Anfragen sichern	112
4.3.2	Eine eigene Anmeldeseite erstellen	115
4.3.3	Abmelden	118
4.3.4	CSRF-Angriffe verhindern	118
4.4	Den Benutzer ermitteln	120
4.5	Zusammenfassung	122
5	Mit Konfigurationseigenschaften arbeiten	123
5.1	Automatische Konfiguration optimieren	124
5.1.1	Die Umgebungsabstraktion von Spring verstehen	124
5.1.2	Eine Datenquelle konfigurieren	126
5.1.3	Den eingebetteten Server konfigurieren	128
5.1.4	Protokollieren konfigurieren	129
5.1.5	Spezielle Eigenschaftswerte verwenden	130
5.2	Eigene Konfigurationseigenschaften erzeugen	131
5.2.1	Holder für Konfigurationseigenschaften definieren	134
5.2.2	Metadaten von Konfigurationseigenschaften deklarieren	136
5.3	Konfigurieren mit Profilen	139
5.3.1	Profilspezifische Eigenschaften definieren	140

5.3.2	Profile aktivieren	141
5.3.3	Beans mit Profilen bedingt erstellen	142
5.4	Zusammenfassung	144
6	REST-Dienste erstellen.....	147
6.1	RESTful Controller programmieren	148
6.1.1	Daten vom Server abrufen.....	150
6.1.2	Daten an den Server senden	155
6.1.3	Daten auf dem Server aktualisieren.....	156
6.1.4	Daten vom Server löschen.....	159
6.2	Hypermedia aktivieren.....	160
6.2.1	Hyperlinks hinzufügen	162
6.2.2	Ressourcenassemble.....	165
6.2.3	Eingebettete Beziehungen benennen	169
6.3	Datengestützte Dienste aktivieren	171
6.3.1	Ressourcenpfade und Beziehungsnamen anpassen	173
6.3.2	Paging und Sortieren	175
6.3.3	Benutzerdefinierte Endpunkte hinzufügen	177
6.3.4	Benutzerdefinierte Hyperlinks zu Spring-Data-Endpunkten hinzufügen ..	179
6.4	Zusammenfassung	180
7	REST-Dienste konsumieren.....	181
7.1	REST-Endpunkte mit RestTemplate konsumieren.....	182
7.1.1	Ressourcen mit GET abrufen.....	184
7.1.2	Ressourcen mit PUT senden	185
7.1.3	Ressourcen mit DELETE löschen	186
7.1.4	Ressourcendaten per POST senden	186
7.2	Mit Traverson in REST APIs navigieren	187
7.3	Zusammenfassung	189
8	Nachrichten asynchron senden	191
8.1	Nachrichten mit JMS senden	192
8.1.1	JMS einrichten	192
8.1.2	Nachrichten mit JmsTemplate senden	194
8.1.3	JMS-Nachrichten empfangen.....	202
8.2	Mit RabbitMQ und AMQP arbeiten	206
8.2.1	RabbitMQ zu Spring hinzufügen	207
8.2.2	Nachrichten mit RabbitTemplate senden.....	208
8.2.3	Nachrichten von RabbitMQ empfangen	212
8.3	Messaging mit Kafka	217
8.3.1	Spring für Kafka-Messaging einrichten.....	218
8.3.2	Nachrichten mit KafkaTemplate senden	219
8.3.3	Kafka-Listener schreiben	221
8.4	Zusammenfassung	222

9 Spring integrieren	223
9.1 Einen einfachen Integrationsfluss deklarieren	224
9.1.1 Integrationsflüsse mit XML definieren	225
9.1.2 Integrationsflüsse in Java konfigurieren	227
9.1.3 Die DSL-Konfiguration von Spring Integration verwenden	229
9.2 Die Landschaft von Spring Integration im Überblick	231
9.2.1 Nachrichtenkanäle	232
9.2.2 Filter	233
9.2.3 Transformer	234
9.2.4 Router	236
9.2.5 Splitter	237
9.2.6 Dienstaktivatoren	240
9.2.7 Gateways	242
9.2.8 Kanaladapter	243
9.2.9 Endpunktmodule	245
9.3 Einen E-Mail-Integrationsfluss erstellen	246
9.4 Zusammenfassung	252
10 Einführung in Reactor	255
10.1 Reaktive Programmierung verstehen	256
10.1.1 Reactive Streams definieren	257
10.2 Erste Schritte mit Reactor	260
10.2.1 Reaktive Datenflüsse grafisch darstellen	261
10.2.2 Reactor-Abhängigkeiten hinzufügen	262
10.3 Allgemeine reaktive Operationen anwenden	263
10.3.1 Reaktive Typen erstellen	264
10.3.2 Reaktive Typen kombinieren	268
10.3.3 Reaktive Streams transformieren und filtern	272
10.3.4 Logische Operationen auf reaktiven Typen ausführen	281
10.4 Zusammenfassung	283
11 Reaktive APIs entwickeln	285
11.1 Mit Spring WebFlux arbeiten	285
11.1.1 Einführung in Spring WebFlux	287
11.1.2 Reaktive Controller schreiben	288
11.2 Funktionale Anfrage-Handler definieren	293
11.3 Reaktive Controller testen	296
11.3.1 GET-Anfragen testen	296
11.3.2 POST-Anfragen testen	299
11.3.3 Mit einem Live-Server testen	300
11.4 REST APIs reaktiv konsumieren	301
11.4.1 Ressourcen mit GET-Anfragen abrufen	302
11.4.2 Ressourcen senden	304
11.4.3 Ressourcen löschen	305
11.4.4 Fehler behandeln	306
11.4.5 Anfragen vermitteln	308

11.5 Reaktive Web-APIs sichern	309
11.5.1 Reaktive Websicherheit konfigurieren	310
11.5.2 Einen reaktiven Dienst für Benutzerdetails konfigurieren	311
11.6 Zusammenfassung	313
12 Daten reaktiv persistent speichern	315
12.1 Die reaktive Geschichte von Spring Data	316
12.1.1 Reaktives Spring Data auf den Punkt gebracht	316
12.1.2 Zwischen reaktiven und nichtreaktiven Typen konvertieren	317
12.1.3 Reaktive Repositories entwickeln	319
12.2 Mit reaktiven Cassandra-Repositories arbeiten	319
12.2.1 Spring Data Cassandra aktivieren	320
12.2.2 Cassandra-Datenmodellierung verstehen	323
12.2.3 Domäentypen für Cassandra-Persistenz abbilden	323
12.2.4 Reaktive Cassandra-Repositories programmieren	329
12.3 Reaktive MongoDB-Repositories programmieren	332
12.3.1 Spring Data MongoDB aktivieren	332
12.3.2 Domäentypen auf Dokumente abbilden	334
12.3.3 Reaktive MongoDB-Repository-Schnittstellen schreiben	338
12.4 Zusammenfassung	341
13 Service-Discovery	345
13.1 Denken in Microservices	346
13.2 Eine Dienstregistrierung einrichten	348
13.2.1 Eureka konfigurieren	352
13.2.2 Eureka skalieren	355
13.3 Dienste registrieren und entdecken	357
13.3.1 Eureka-Clienteigenschaften konfigurieren	358
13.3.2 Dienste konsumieren	359
13.4 Zusammenfassung	365
14 Konfiguration verwalten	367
14.1 Konfiguration teilen	368
14.2 Config Server ausführen	369
14.2.1 Config Server aktivieren	370
14.2.2 Das Konfigurations-Repository füllen	373
14.3 Gemeinsame Konfigurationen konsumieren	376
14.4 Anwendungs- und profilspezifische Eigenschaften bereitstellen	378
14.4.1 Anwendungsspezifische Eigenschaften bereitstellen	378
14.4.2 Eigenschaften von Profilen bereitstellen	379
14.5 Konfigurationseigenschaften geheim halten	381
14.5.1 Eigenschaften in Git verschlüsseln	382
14.5.2 Geheimnisse in Vault speichern	385
14.6 Konfigurationseigenschaften im laufenden Betrieb aktualisieren	390
14.6.1 Konfigurationseigenschaften manuell aktualisieren	391
14.6.2 Konfigurationseigenschaften automatisch aktualisieren	393

14.7 Zusammenfassung	401
15 Fehler und Latenzzeiten behandeln	403
15.1 Trennschalter im Überblick	403
15.2 Trennschalter deklarieren	405
15.2.1 Latenz reduzieren	408
15.2.2 Schwellenwerte für Trennschalter verwalten	409
15.3 Fehler überwachen	411
15.3.1 Das Hystrix-Dashboard – eine Einführung	412
15.3.2 Hystrix-Threadpools	415
15.4 Hystrix-Streams aggregieren	416
15.5 Zusammenfassung	418
16 Mit Spring Boot Actuator arbeiten	421
16.1 Actuator im Überblick	421
16.1.1 Den Basispfad von Actuator konfigurieren	423
16.1.2 Actuator-Endpunkte aktivieren und deaktivieren	424
16.2 Actuator-Endpunkte konsumieren	425
16.2.1 Wichtige Anwendungsinformationen abrufen	426
16.2.2 Konfigurationsdetails ansehen	429
16.2.3 Anwendungsaktivität anzeigen	437
16.2.4 Laufzeit-Metriken erfassen	440
16.3 Actuator anpassen	443
16.3.1 Informationen zum Endpunkt /info beisteuern	443
16.3.2 Benutzerdefinierte Zustandsindikatoren definieren	448
16.3.3 Benutzerdefinierte Metriken registrieren	449
16.3.4 Benutzerdefinierte Endpunkte erstellen	451
16.4 Actuator sichern	454
16.5 Zusammenfassung	456
17 Spring verwalten	457
17.1 Spring Boot Admin verwenden	457
17.1.1 Einen Admin-Server erstellen	458
17.1.2 Admin-Clients registrieren	460
17.2 Admin-Server im Detail	464
17.2.1 Integritätsdaten und allgemeine Anwendungsinformationen anzeigen	465
17.2.2 Schlüsselmetriken überwachen	467
17.2.3 Umgebungseigenschaften untersuchen	468
17.2.4 Protokollierungsstufen anzeigen und festlegen	469
17.2.5 Threads überwachen	470
17.2.6 HTTP-Anfragen verfolgen	471
17.3 Den Admin-Server sichern	473
17.3.1 Anmelden beim Admin-Server aktivieren	473
17.3.2 Beim Actuator authentifizieren	474
17.4 Zusammenfassung	475

18 Spring mit JMX überwachen	477
18.1 Mit Actuator-MBeans arbeiten.....	477
18.2 Eigene MBeans erstellen	480
18.3 Benachrichtigungen senden	482
18.4 Zusammenfassung	483
19 Spring bereitstellen	485
19.1 Bereitstellungsoptionen abwägen	486
19.2 WAR-Dateien erstellen und bereitstellen	487
19.3 JAR-Dateien zu Cloud Foundry verschieben	489
19.4 Spring Boot in einem Docker-Container ausführen.....	492
19.5 Der Weg ist das Ziel	496
19.6 Zusammenfassung	497
A Bootstrapping von Spring-Anwendungen	499
A.1 Ein Projekt mit Spring Tool Suite initialisieren.....	499
A.2 Ein Projekt mit IntelliJ IDEA initialisieren.....	503
A.3 Ein Projekt mit NetBeans initialisieren	507
A.4 Ein Projekt unter start.spring.io initialisieren.....	511
A.5 Ein Projekt von der Befehlszeile initialisieren.....	515
A.5.1 curl und die Initializr API	515
A.5.2 Befehlszeiloberfläche von Spring Boot.....	517
A.6 Spring-Anwendungen mit einem Meta-Framework erstellen.....	519
A.7 Projekte erstellen und ausführen	519
Stichwortverzeichnis	521

Vorwort

Nachdem ich fast 15 Jahre mit Spring gearbeitet und mehrere Ausgaben dieses Buches geschrieben habe (ganz zu schweigen von „*Spring Boot in Action*“), sollte man meinen, dass sich kaum noch etwas Aufregendes und Neues über Spring sagen lässt, wenn es um das Vorwort für dieses Buch geht. Aber wie so oft sieht es in der Realität ganz anders aus!

Jedes einzelne Release von Spring, Spring Boot und allen anderen Projekten im Spring-Ökosystem schafft neue erstaunliche Möglichkeiten, die den Spaß an der Entwicklung von Anwendungen wieder aufleben lassen. Mit dem Release 5.0 von Spring und dem Release 2.0 von Spring Boot gibt es so viel mehr Spring zu genießen, dass es ein Kinderspiel war, eine weitere Ausgabe von *Spring im Einsatz* zu schreiben.

Das Großartige an Spring 5 ist die Unterstützung für reaktive Programmierung, unter anderem für Spring WebFlux, ein brandneues reaktives Web-Framework, dessen Programmiermodell sich an Spring MVC orientiert. Damit können Entwickler Webanwendungen schaffen, die sich besser skalieren lassen und weniger Threads effektiver nutzen. In Richtung des Backends einer Spring-Anwendung erlaubt es die neueste Edition von Spring Data, reaktive, nicht blockierende Daten-Repositories aufzubauen. Und alles dies baut auf Project Reactor auf, einer Java-Bibliothek für das Arbeiten mit reaktiven Typen.

Zusätzlich zu den neuen reaktiven Programmierfeatures von Spring 5 bietet Spring Boot 2 nun sogar mehr Unterstützung als je zuvor für die Autokonfiguration sowie einen vollständig neu konzipierten Actuator, mit dem sich eine laufende Anwendung inspizieren und manipulieren lässt.

Da Entwickler zudem ihre monolithischen Anwendungen in diskrete Microservices aufteilen wollen, bietet Spring Cloud Einrichtungen, die es erleichtern, Microservices zu konfigurieren, zu erkennen und sie widerstandsfähiger gegen Ausfälle zu machen.

Ich freue mich, sagen zu können, dass diese fünfte Ausgabe von *Spring im Einsatz* – hier vorliegend als Übersetzung in der dritten Auflage – alle diese und noch mehr Themen abdeckt! Wenn Sie ein erfahrener Veteran von Spring sind, wird *Spring im Einsatz* Ihr Leitfaden für alles Neue sein, das Spring zu bieten hat. Wenn Sie andererseits neu in Spring einsteigen, dann gibt es keinen besseren Zeitpunkt als jetzt, um richtig loszulegen. Die ersten Kapitel bringen Sie im Handumdrehen an den Start!

Die 15 Jahre Arbeit mit Spring sind eine spannende Zeit gewesen. Und da nun diese Edition von *Spring im Einsatz* vor Ihnen liegt, bin ich versessen darauf, diese Begeisterung mit Ihnen zu teilen!

Danksagungen

Zu den erstaunlichsten Dingen bei Spring und Spring Boot ist zu nennen, dass sie automatisch alle grundlegenden Installationen für eine Anwendung bereitstellen, wodurch Sie sich als Entwickler vorrangig auf die Logik konzentrieren können, die Ihre Anwendung im Speziellen ausmacht. Leider gibt es keine solchen magischen Hilfsmittel, um ein Buch zu schreiben. Oder etwa doch?

Bei Manning haben mehrere Leute ihre magischen Kräfte entfaltet, um sicherzustellen, dass dieses Buch das Beste wird, was möglich ist. Vielen Dank insbesondere an Jenny Stout, meine Entwicklungsredakteurin, und das Produktteam, darunter Projektleiterin Janet Vail, die Copyeditoren Andy Carroll und Frances Buran sowie das Korrektorat mit Katie Tenant und Melody Dolab. Dank auch dem Fachlektor Joshua White, der gründlich und hilfreich war.

In dieser Zeit haben wir auch Feedback von mehreren Gutachtern erhalten, die dafür gesorgt haben, den Kurs zu halten und die richtigen Themen abzudecken. Dafür danke ich Andrea Barisone, Arnaldo Ayala, Bill Fly, Colin Joyce, Daniel Vaughan, David Witherspoon, Eddu Melendez, Iain Campbell, Jetro Coenradie, John Gunvaldson, Markus Matzker, Nick Rakochy, Nusry Firdousi, Piotr Kafel, Raphael Villela, Riccardo Noviello, Sergio Fernandez Gonzalez, Sergiy Pylypets, Thiago Presa, Thorsten Weber, Waldemar Modzelewski, Yagiz Erkan und Željko Troglić.

Wie immer gäbe es absolut keinen Grund, dieses Buch zu schreiben, wenn da nicht die erstaunliche Arbeit des Spring-Entwicklerteams wäre. Ich kann nur darüber staunen, was es geschaffen hat und wie wir den Entwicklungsstil von Software immer wieder verändern.

Ein großer Dank geht an meine Mitstreiter auf der No Fluff/Just Stuff-Tour. Ich lerne weiterhin so viel von jedem von euch! Besonders danken möchte ich Brian Sletten, Nate Schutta und Ken Kousen für die Gespräche und E-Mails über Spring, die zur Gestaltung dieses Buches beigetragen haben.

Nochmals vielen Dank an die Phönizier. Ihr wisst, was ihr getan habt.

An meine wundervolle Frau Raymie, die Liebe meines Lebens, meinen süßesten Traum und meine Inspiration gerichtet: Danke für dein Engagement und dafür, dass du dich mit einem weiteren Buchprojekt abgefunden hast. Und an meine süßen und wundervollen Mädchen, Maisy und Madi: Ich bin so stolz auf euch und auf die erstaunlichen jungen Damen, die ihr einmal sein werdet. Ich liebe euch alle mehr, als ihr es euch vorstellen könnt oder ich es auszudrücken vermag.

Über dieses Buch

Spring im Einsatz soll Sie in die Lage versetzen, erstaunliche Anwendungen mit dem Spring Framework, Spring Boot und einer breiten Palette von Ergänzungstools des Spring-Ökosystems zu erstellen. Zunächst erfahren Sie, wie Sie webbasierte, datenbankgestützte Java-Anwendungen mit Spring und Spring Boot entwickeln. Anschließend geht es über die Grundlagen hinaus und es wird gezeigt, wie Sie die Integration mit anderen Anwendungen realisieren, mit reaktiven Typen programmieren und dann eine Anwendung in diskrete Microservices aufteilen. Schließlich wird erörtert, wie Sie eine Anwendung für die Bereitstellung fit machen. Obwohl alle Projekte im Spring-Ökosystem eine ausgezeichnete Dokumentation bieten, gibt Ihnen dieses Buch etwas, was Sie in den Referenzdokumentationen nicht finden: einen praktischen, projektgetriebenen Leitfaden, um die Elemente von Spring im Rahmen einer realen Anwendung zusammenzubringen.

Wer dieses Buch lesen sollte

Die vorliegende Ausgabe von *Spring im Einsatz*, richtet sich an Java-Entwickler, die erste Schritte mit Spring Boot und dem Spring Framework unternehmen möchten, sowie an erfahrene Spring-Entwickler, die über die Grundlagen hinausgehen und die neuesten Features von Spring kennenlernen wollen.

Wie dieses Buch organisiert ist: eine Roadmap

Das Buch umfasst 19 Kapitel, die in fünf Teile gegliedert sind. **Teil I** befasst sich mit den grundlegenden Themen für das Erstellen von Anwendungen:

- *Kapitel 1* führt Spring und Spring Boot ein und zeigt, wie Sie ein Spring-Projekt initialisieren. In diesem Kapitel unternehmen Sie die ersten Schritte und erstellen eine Spring-Anwendung, die Sie im weiteren Verlauf des Buches erweitern und vervollständigen werden.
- *Kapitel 2* erläutert, wie Sie die Web-Ebene einer Anwendung mit Spring MVC aufbauen. In diesem Kapitel erstellen Sie Controller, die Webanfragen behandeln, und Views, die Informationen im Webbrowser darstellen.
- *Kapitel 3* beschäftigt sich mit dem Backend einer Spring-Anwendung, wo die Daten in einer relationalen Datenbank persistent gespeichert werden.
- In *Kapitel 4* nutzen Sie Spring Security, um Benutzer zu authentifizieren und nicht autorisierten Zugriff auf eine Anwendung zu verhindern.
- *Kapitel 5* macht deutlich, wie Sie eine Spring-Anwendung mit Spring-Boot-Konfigurations-eigenschaften konfigurieren. Außerdem lernen Sie, wie Sie eine Konfiguration mithilfe von Profilen selektiv anwenden.

Die Themen in **Teil II** helfen Ihnen, wenn Sie Ihre Spring-Anwendung mit anderen Anwendungen integrieren:

- *Kapitel 6* erweitert die in Kapitel 2 begonnene Diskussion zu Spring MVC und zeigt, wie sich REST APIs in Spring schreiben lassen.
- *Kapitel 7* tauscht die Rollen gegenüber Kapitel 6 und zeigt, wie eine Spring-Anwendung eine REST API konsumieren kann.
- *Kapitel 8* befasst sich mit asynchroner Kommunikation, damit eine Spring-Anwendung per Java Message Service, RabbitMQ oder Kafka Nachrichten sowohl senden als auch empfangen kann.
- In *Kapitel 9* geht es um deklarative Anwendungsintegration mit dem Spring-Integration-Projekt.

Teil III ist der neuen Unterstützung für reaktive Programmierung in Spring gewidmet:

- *Kapitel 10* stellt Project Reactor vor, die reaktive Programmierbibliothek, die die reaktiven Features von Spring 5 untermauert.
- *Kapitel 11* beschäftigt sich noch einmal mit der REST-API-Entwicklung und führt Spring WebFlux ein, ein neues Web-Framework, das sich stark an Spring MVC orientiert, dabei aber ein neues reaktives Modell für die Web-Entwicklung bietet.
- In *Kapitel 12* werfen wir einen Blick darauf, wie sich reaktive Datenpersistenz mit Spring Data programmieren lässt, um Daten in und aus den Datenbanken Cassandra und Mongo zu schreiben und zu lesen.

Teil IV zerlegt das monolithische Anwendungsmodell und führt Sie in die Entwicklung mit Spring Cloud und Microservices ein:

- *Kapitel 13* befasst sich mit dem Erkennen von Diensten, wobei Sie Spring mit der Eureka-Registrierung von Netflix verwenden, um Spring-basierte Microservices sowohl zu registrieren als auch zu erkennen.
- In *Kapitel 14* zentralisieren Sie die Anwendungskonfiguration auf einem Konfigurationsserver, der die Konfiguration für mehrere Microservices gemeinsam nutzt.
- *Kapitel 15* führt das Trennschalter-Muster (Circuit Breaker) mit Hystrix ein, das es ermöglicht, Microservices für den Fehlerfall robuster zu machen.

In **Teil V** bereiten Sie eine Anwendung für die Produktion vor und lernen, wie Sie sie bereitstellen:

- *Kapitel 16* stellt den Spring Boot Actuator vor, eine Erweiterung zu Spring Boot, mit der sich die Interna einer laufenden Spring-Anwendung als REST-Endpunkte zugänglich machen lassen.
- In *Kapitel 17* sehen Sie, wie Sie mit Spring Boot Admin eine benutzerfreundliche browserbasierte administrative Anwendung auf Actuator aufsetzen können.
- *Kapitel 18* erläutert, wie sich Spring-Beans als JMX MBeans zugänglich machen und konsumieren lassen.
- Abschließend zeigt *Kapitel 19*, wie Sie Ihre Spring-Anwendung in den verschiedensten Produktionsumgebungen bereitstellen.

Im Allgemeinen sollten Entwickler, die in Spring einsteigen, mit Kapitel 1 beginnen und alle Kapitel nacheinander durcharbeiten. Erfahrene Spring-Entwickler ziehen es vielleicht vor, gleich mit dem Thema zu beginnen, das sie vorrangig interessiert. Es sei aber darauf

hingewiesen, dass jedes Kapitel auf dem vorhergehenden aufbaut, sodass möglicherweise der Kontext unklar ist, wenn Sie gleich in der Mitte des Buches einsteigen.

Über den Code

Dieses Buch enthält viele Beispiele im Quellcode sowohl in nummerierten Listings als auch in Form von Codefragmenten, die in den laufenden Text eingefügt sind. In beiden Fällen ist der Quellcode in **Einer-Schreibmaschinenschrift-wie-dieser** formatiert, um ihn vom normalen Text zu trennen. Manchmal ist der Code auch **fett** gedruckt, um ihn von Code abzuheben, der sich gegenüber vorherigen Schritten im Kapitel geändert hat, beispielsweise wenn ein neues Feature zu einer schon vorhandenen Codezeile hinzukommt.

Der Quellcode zu den Beispielen in diesem Buch steht auf der Website des Verlages der Originalausgabe unter www.manning.com/books/spring-in-action-fifth-edition sowie im GitHub-Konto des Autors unter github.com/habuma/spring-in-action-5-samples zum Download bereit.

Buchforum

Beim Kauf dieser Ausgabe von *Spring im Einsatz* erhalten Sie kostenfreien Zugriff auf ein privates Web-Forum unter Leitung von Manning Publications, in dem Sie Kommentare zum Buch abgeben, technische Fragen stellen und Hilfe vom Autor und von anderen Benutzern erhalten können. Um auf das Forum zuzugreifen, besuchen Sie <https://forums.manning.com/forums/spring-in-action-fifth-edition>. Mehr über die Foren von Manning und die Verhaltensregeln erfahren Sie auch unter <https://forums.manning.com/forums/about>.

Manning engagiert sich für die Leser, um ihnen einen Treffpunkt zu bieten, an dem ein sinnvoller Dialog zwischen Lesern untereinander und zwischen dem Leser und dem Autor stattfinden kann. Es handelt sich dabei nicht um eine Verpflichtung dem Autor gegenüber, in einem bestimmten Umfang mitzuwirken, wobei der Beitrag zum Forum freiwillig (und unbezahlt) bleibt. Stellen Sie doch dem Autor herausfordernde Fragen, damit sein Interesse nicht verloren geht! Das Forum und die Archive früherer Diskussionen sind über Website des Verlags zugänglich, solange das Buch lieferbar ist.

Andere Online-Quellen

Brauchen Sie zusätzliche Hilfe?

- Die Spring-Website bietet unter <https://spring.io/guides> mehrere nützliche Leitfäden (von denen der Autor selbst einige geschrieben hat).
- Die Tags für Spring auf StackOverflow (<https://stackoverflow.com/questions/tagged/spring>) und Spring Boot auf StackOverflow sind empfehlenswerte Orte rund um das Thema Spring, um Fragen zu stellen und anderen zu helfen. Gleichzeitig ist es ein guter Weg, um mehr über Spring zu lernen, wenn man anderen bei der Beantwortung ihrer Spring-Fragen hilft!

Über den Autor

CRAIG WALLS ist leitender Ingenieur bei Pivotal. Er ist ein eifriger Förderer des Spring Frameworks, ist häufig in lokalen Benutzergruppen und Konferenzen anzutreffen und schreibt über Spring. Wenn er nicht gerade mit Code um sich wirft, plant Craig seine nächste Reise nach Disney World oder Disneyland und verbringt so viel Zeit wie möglich mit seiner Frau, seinen beiden Töchtern, zwei Vögeln und drei Hunden.

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?

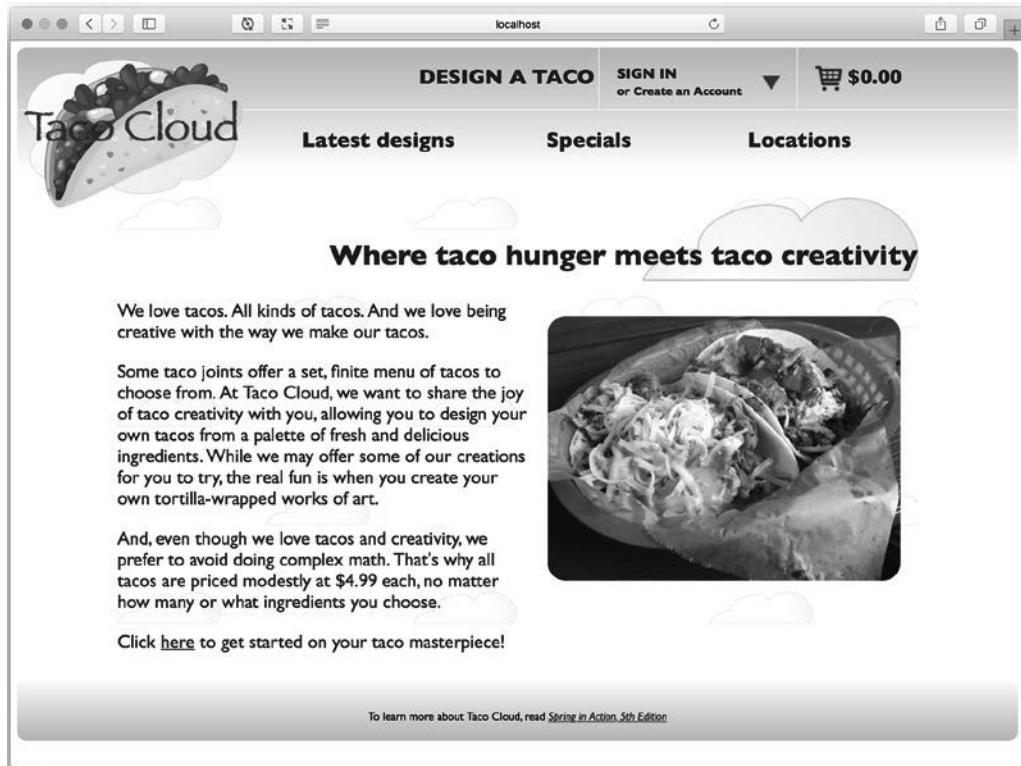


Bild 6.1 Die neue Taco-Cloud-Startseite

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-basierte¹ 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*
<code>@GetMapping</code>	HTTP-GET-Anfragen	Ressourcendaten lesen
<code>@PostMapping</code>	HTTP-POST-Anfragen	Eine Ressource erstellen
<code>@PutMapping</code>	HTTP-PUT-Anfragen	Eine Ressource aktualisieren
<code>@PatchMapping</code>	HTTP-PATCH-Anfragen	Eine Ressource aktualisieren
<code>@DeleteMapping</code>	HTTP-DELETE-Anfragen	Eine Ressource löschen
<code>@RequestMapping</code>	Universelle Anfragebehandlung; HTTP-Methode im <code>method</code> -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 Ergebnis 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 Ersetzungsoperationen 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, inspiert 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 Domänen Daten 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 Domänen Daten 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) {}
}
```

Mittlerweile 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 über gibt 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": "TMT0", "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/LETCE" }
            }
          },
          {
            "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 Ressourcenassembleuren, 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 Domäentypen 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 API² 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. Statt dessen 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-Domäentyp. 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 Domänenotypen `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 Ressourcen-links 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 instanzieren. 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 abgleitet 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 instanzieren, 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.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 Domäntyp 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 refaktorieren 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/FLT0"
        }
      }
    }
  }
}
```

```
        "href" : "http://localhost:8080/ingredients/FLT0
    }
}
},
...
]
},
"_links" : {
  "self" : {
    "href" : "http://localhost:8080/ingredients"
  },
  "profile" : {
    "href" : "http://localhost:8080/profile/ingredients"
  }
}
```

Wow! Indem Sie lediglich eine Abhangigkeit in Ihren Build einfugen, bekommen Sie nicht nur einen Endpunkt fur Zutaten, sondern die zurckgegebenen Ressourcen enthalten auch Hyperlinks! Wenn Sie sich als Client dieser API ausgeben, knnen Sie auch per `curl` dem `self`-Link fr die Weizen-Tortilla (Flour Tortilla) folgen:

```
$ curl http://localhost:8080/ingredients/FLT0
{
  "name" : "Flour Tortilla",
  "type" : "WRAP",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/ingredients/FLT0"
    },
    "ingredient" : {
      "href" : "http://localhost:8080/ingredients/FLT0"
    }
  }
}
```

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 `createdAt` 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.

Stichwortverzeichnis

Symbol

@{} 41
<dependency>
- Sicherheit 91
\${} 130
@AllowFiltering 331
- where-Klausel 331
@AuthenticationPrincipal 120
@Autowired 66
@Bean annotation 5
/beans 424, 429
@ComponentScan 15
@ConditionalOnClass 431
@ConditionalOnMissingBean 431
/conditions 424
/configprops 429
@Configuration (Annotation) 5
@ConfigurationProperties 131, 133, 214, 247
@Controller 18
@CrossOrigin 153
_csrf 119
@Data 33, 34
@DeleteMapping 159
@Digits 51
@Document 334
@EnableAdminServer 458
@EnableAutoConfiguration 15
@EnableConfigServer 371
@EnableEurekaServer 350
@EnableFeignClients 363
@EnableHystrix 406
@EnableHystrixDashboard 412
@EnableWebFluxSecurity 311
@EnableWebSecurity 310
@Endpoint 453
@Entity 324
/env 431
@FeignClient 364
@Field 334
@GeneratedValue 84
@GetMapping 18, 36, 149, 155
@Header 225
/health 424
/heapdump 437
/httptrace 437, 471
@HystrixCommand 405
- commandProperties (Attribut) 408
@Id 83, 334
@ImportResource 227
@InboundChannelAdapter 243
/info 424
- Informationen beisteuern 443
@JmsListener 204, 205, 216
@JmxEndpoint 453
@KafkaListener 221
_links 162
@LoadBalanced 360
/loggers 437
@ManagedAttribute 480, 481
@ManagedOperation 480, 481
@ManagedResource 480
@ManyToMany 84
@ManyToOne 120
/mappings 429, 434
@Max 136
@MessagingGateway 225
/metrics 440
@Min 136
@ModelAttribute 76
! (Negation) 143
@NoArgsConstructor 83
@NotBlank 50
@NotNull 49
@PathVariable 364
@Pattern 51
@PostMapping 43, 149, 155
@PrePersist 84
@Repository 66

- @RepositoryRestController 177, 178
 - @RequestBody 156
 - @RequestMapping 36, 153
 - @RequiredArgsConstructor 83
 - @RestController 152
 - Annotation entfernen 171
 - @RestResource 175
 - @Router 236
 - @RunWith 16, 300
 - @Service 107, 481
 - @ServiceActivator annotation 228, 240
 - @Size 49
 - @Slf4j 36
 - @SpringBootApplication 15
 - @SpringBootConfiguration 15
 - @SpringBootTest 17, 300
 - @Table 85, 324
 - /threaddump 438, 470
 - @Transformer 235
 - @UserDefinedType 326
 - @Valid 52
 - @Validated 136
 - @WebEndpoint 453
 - @WebMvcTest 21, 56
- A**
- Abilden, reaktive Daten 275
 - Abbilder 492
 - Abhängigkeiten 509
 - Hinzufügen 34
 - Spring Cloud Stream 397
 - Spring Data Reactive Cassandra 321
 - Spring WebFlux 287
 - Ablaufdatum 50
 - Abmelden, logout() 118
 - AbstractMessageRouter 237
 - AbstractRepositoryEventListener 481
 - abstrakte Datentypen 326
 - Accept 153
 - access() 113
 - ActiveMQQueue 197
 - Actuator *siehe* Spring Boot Actuator
 - Endpunkte 423
 - Endpunkte, Übersicht 423
 - Micrometer 449
 - Actuator-Endpunkte *siehe* Endpunkte
 - additional-spring-configuration-metadata.json 137
 - addNote() 452
 - addScript() 124
 - addScripts() 124
 - addViewControllers() 55, 116
 - Advanced Message Queueing Protocol (AMQP) 191
 - Algorithmen
 - Luhn- 51
 - all() 281
 - AllIgnoresCase 89
 - AllIgnoringCase 89
 - AMQP (Advanced Message Queueing Protocol) 191
 - Spring Cloud Bus 400
 - AmqpHeaderConverter 211
 - AmqpTemplate (Schnittstelle) 208
 - and() 115
 - andRoute() 294
 - Anfragen
 - Accept 153
 - sichern 112
 - Angriffe
 - CSRF 118
 - Anmelden
 - Abmelden 118
 - Anmeldeseite erstellen 115
 - Pfad 116
 - Annotationen
 - @AllowFiltering 331
 - @AuthenticationPrincipal 120
 - @Autowired 66
 - @ConditionalOnClass 431
 - @ConditionalOnMissingBean 431
 - @ConfigurationProperties 131, 133, 214, 247
 - @CrossOrigin 153
 - @DeleteMapping 159
 - @Document 334
 - @EnableAdminServer 458
 - @EnableConfigServer 371
 - @EnableEurekaServer 350
 - @EnableFeignClients 363
 - @EnableHystrix 406
 - @EnableHystrixDashboard 412
 - @EnableWebFluxSecurity 311
 - @EnableWebSecurity 310
 - @Endpoint 453
 - @Entity 324
 - @FeignClient 364
 - @Field 334
 - @GeneratedValue 84
 - @GetMapping 36, 149, 155
 - @Header 225

- @HystrixCommand 405
- @Id 83, 334
- @ImportResource 227
- @InboundChannelAdapter 243
- @JmsListener 204, 205, 216
- @JmxEndpoint 453
- @KafkaListener 221
- Komponentensuche 152
- @LoadBalanced 360
- @ManagedAttribute 480, 481
- @ManagedOperation 480, 481
- @ManagedResource 480
- @ManyToMany 84
- @ManyToOne 120
- @Max 136
- @MessagingGateway 225
- @Min 136
- @ModelAttribute 76
- @NoArgsConstructor 83
- @NotNull 49
- @PathVariable 364
- @Pattern 51
- @PostMapping 43, 149, 155
- @PrePersist 84
- @Repository 66
- @RepositoryRestController 177, 178
- @RequestBody 156
- @RequestMapping 36, 153
- @RequiredArgsConstructor 83
- @RestController 152
- @RestResource 175
- @Router 236
- @RunWith 300
- @Service 481
- @Size 49
- @SpringBootTest 300
- @Table 85, 324
- @Transformer 235
- @UserDefinedType 326
- @Valid 52
- @Validated 136
- @WebEndpoint 453
- @WebMvcTest 56
- antMatchers() 311
- Anwendungen
 - Bootstrapping 499
 - Cloud-native 347
 - hochfahren 15
 - Projekte mit Spring Tool Suite initialisieren 499
- Anwendungskontext 4
- any() 281
- Anzeigen, Informationen 31
- Apache Kafka 217
- ApiProperties 251
- APIs
 - Controller, reaktive 288
 - reaktive 285
 - Sichern reaktiver Web- 309
- applicationConfig 433
- applicationName 516
- application.yml 126
- artifactId 516
- asLink() 189
- Asynchrones Messaging 191
 - JMS 192
 - JMS einrichten 192
 - Kafka 217
 - Kafka einrichten 218
 - Kafka-Listener 221
 - Nachrichten empfangen 202
 - Nachrichten mit KafkaTemplate senden 219
 - Nachrichten mit RabbitMQ empfangen 212
 - Nachrichten senden mit RabbitTemplate 208
 - RabbitMQ 206
- AtomicInteger 243
- Attribute
 - commandProperties 408
 - defaultRequestChannel 225
 - directory 226
 - Servlet-Anfrage- 39
 - th:each 39
 - th:errors 53
 - th:text 39
 - Thymeleaf 39
 - userPassword 101
 - webEnvironment 300
- Auflistungen
 - Reaktive Typen erstellen 265
- Ausnahmen
 - EmptyResultDataAccessException 159
 - RestClientException 407
 - UsernameNotFoundException 106
- AuthenticationManagerBuilder 96
- Authentifizierung
 - Benutzerauthentifizierung anpassen 104
 - Dienst für Benutzerdetails 106
 - ldapAuthentication() 100
- Autokonfiguration 6, 430

- bedingte 431
- Datenquelle 126
- Eigenschaftswerte, spezielle 130
- Konfiguration eines eingebetteten Servers 128
- Optimieren 124
- Protokollierung 129
- Umgebungsabstraktion 124
- Autovervollständigung 138
- Autowiring 5
- availableTags 441

- B**
- Backpressure 257
- baseDir 516
- Base Url 8, 502
- Basispfad
 - Actuator 423
 - Anpassen 172
 - Controller 177
- bcrypt 99
- BCryptPasswordEncoder 99
- Beans 4
 - bedingt erstellen mit Profilen 142
 - Bericht zum Wiring 429
 - CommandLineRunner 143
 - InventoryService 5
 - MongoTemplate 431
 - ProductService 5
 - RabbitTemplate 207
 - Verknüpfen 124
- Befehle
 - spring init 517
 - tar 517
- Befehlszeile
 - curl 154
 - curl und Initializr API 515
 - HTTPPie 154
 - Profile aktivieren 141
 - Projekte initialisieren 515, 517
- Benutzerabfragen, Überschreiben
 - standardmäßiger 97
- Benutzerauthentifizierung
 - Anpassen 104
 - Benutzerdomäne und Persistenz 104
 - Benutzer registrieren 109
 - Dienst für Benutzerdetails 106
- benutzerdefinierte Typen 326
- Benutzeroberflächen 457
- Benutzerspeicher
 - JDBC-basierter 97
- Konfigurieren 95
- speicherinterner 96
- Bereitstellung 485
 - JAR-Dateien zu Cloud Foundry verschieben 489
 - Optionen 486
 - Spring Boot in Docker-Container 492
 - WAR-Dateien 487
- Between 88
- Beziehungen
 - benennen 169
- Beziehungsnamen
 - Anpassen 173
- Bibliotheken
 - Feign 363
 - Jackson 78
 - Traverson 187
- Bill of Materials (BOM) 262
- BOM (Bill of Materials) 262
- Bootstrapping
 - Anwendungen 499
 - Apps mit Meta-Framework erstellen 519
 - Projekte erstellen und ausführen 519
 - Projekte mit IntelliJ IDEA initialisieren 503
 - Projekte mit NetBeans initialisieren 507
 - Projekte unter start.spring.io initialisieren 511
 - Projekte von der Befehlszeile initialisieren 515
- bootVersion 516
- Browser, Aktualisierung 24
- build.gradle 488
- BuildInfoContributor 445
- build-info.properties 445
- Build-Spezifikation 12

- C**
- Caching
 - Templates 59
- Cassandra 319
 - aktivieren 320
 - CQL 327
 - cqlsh 327
 - Datenmodellierung 323
 - Domäentypen für Persistenz abbilden 323
 - Eigenschaften 321
 - Programmieren 329
 - Replikation 321
 - Schlüssel 323
 - Single Point of Failure 319
 - Typen, benutzerdefinierte 326
 - where-Klausel 331

- CassandraRepository (Schnittstelle) 330
 - ccExpiration 50
 - ccNumber 50
 - cf
 - bind-service 491
 - create-service 491
 - marketplace 491
 - restage 492
 - channel() 230
 - circuitBreaker.errorThresholdPercentage 410
 - Circuit Breaker Pattern *siehe* Trennschalter
 - circuitBreaker.requestVolumeThreshold 410
 - circuitBreaker.sleepWindowInMilliseconds 410
 - CLI (Command-Line Interface) 515
 - Clients
 - Admin-Clientanwendungen explizit konfigurieren 460
 - Registrieren 460
 - cloud 142
 - Cloud Foundry 142, 485
 - Cloud-native Anwendungen 347
 - Clustering-Schlüssel 323
 - Code
 - imperativer 255
 - reaktiver 255
 - collectList() 280
 - Command Line Interface (CLI) 515
 - CommandLineRunner 143
 - commandProperties 408
 - Component Scanning 5
 - config.client.version 392
 - Config Server 370
 - ausführen 369
 - Release Train 371
 - configuration() 310
 - configure() 95, 487
 - WebSecurityConfigurerAdapter 112
 - configuredLevel 436
 - Container 4
 - ContentTypeDelegatingMessageConverter 211
 - contextSource() 102
 - contribute() 444
 - Controller
 - Basispfad 177
 - Testen 20
 - View- 55
 - Controller-Klasse
 - Erstellen 34
 - ControllerLinkBuilder 163
 - convertAndSend() 195, 198, 209
 - copyToString 298
 - CORS (Cross-Origin Resource Sharing) 153
 - CQL 327
 - (Cassandra Query Language) 321
 - create keyspace (Befehl) 321
 - cqlsh 327
 - createdAt 64
 - createdAt() 84
 - createdDate 176
 - create keyspace (Befehl) 321
 - createMessage() 196
 - CrudRepository 87, 338
 - CrudRepository (Schnittstelle) 330
 - CSRF
 - Token 119
 - CSRF (Cross-Site-Request-Forgery) 118
 - curl 7, 154, 515
- D**
- DataSource 97
 - data.sql 124
 - Dateien
 - additional-spring-configuration-metadata.json 137
 - build.gradle 488
 - build-info.properties 445
 - design.component.ts 155
 - Dockerfile 493
 - Java Cryptography Extensions Unlimited Strength 382
 - Metadaten 137
 - Schlüsselspeicher 382
 - Daten
 - Abrufen vom Server 150
 - Arbeiten mit JdbcTemplate 65
 - Domäne als Entitäten annotieren 82
 - Einfügen mit JdbcTemplate 72
 - Einfügen mit SimpleJdbcTemplate 76
 - JDBC-Repositories definieren 65
 - JPA-Repositories anpassen 87
 - JPA-Repositories deklarieren 86
 - Laden im Voraus 69
 - Lesen/Schreiben mit JDBC 61
 - Löschen vom Server 159
 - Persistenz 64
 - Puffern 278
 - Quelle konfigurieren 126
 - Schema definieren 69
 - Senden an Server 155
 - Speichern mit JdbcTemplate 72

- Spring Data JPA zum Projekt hinzufügen 81
- Zeilen einfügen 68
- Datenbanken
 - eingebettete 333
 - Flapdoodle Embedded MongoDB 333
 - Kennwortverschlüsselung 99
 - Port 27017 494
 - Zeilen einfügen 68
- Datenflüsse, grafisch darstellen 261
- datengestützte Dienste
 - Aktivieren 171
 - Endpunkte, benutzerdefinierte 177
 - Hyperlinks zu Spring-Data-Endpunkten hinzufügen 179
 - Paging und Sortieren 175
 - Ressourcenpfade und Beziehungsnamen anpassen 173
- defaultRequestChannel 225
- defaultSuccessUrl() 117
- delayElements() 269
- delaySubscription() 269
- delete() 186
- deleteById() 159
- deleteNote() 453
- deleteOrder() 159
- dependencies 516
- Dependency Injection (DI) 4
- Deployment *siehe* Bereitstellung
- description 516
- design.component.ts 155
- DesignTacoController 35
- DI (Dependency Injection) 4
- Dienstaktivatoren 240
- Dienste
 - Benutzerdetails 106
 - Konsumieren mit RestTemplate 360
- DirectChannel 232
- directory 226
- disable() 119
- diskSpace 428
- DispatcherServlet 487
- distinct() 275
- Docker
 - Abbilder 492
 - Einstiegspunkte 493
 - Images 492
 - Profile 494
 - Spring Boot in Container ausführen 492
- Dockerfile 493
- docker (Profil) 494
- Domänen, einrichten 33
- doTransform() 249
- DSL, Integrationsflüsse konfigurieren 229
- E**
- Eigenschaften
 - \${} 130
 - applicationConfig 433
 - Cassandra 321
 - circuitBreaker.errorThresholdPercentage 410
 - circuitBreaker.requestVolumeThreshold 410
 - circuitBreaker.sleepWindowInMilliseconds 410
 - config.client.version 392
 - configuredLevel 436
 - createdAt 64
 - createdDate 176
 - encrypt.key 382
 - eureka.client.fetch-registry 353
 - eureka.client.register-with-eureka 353, 462
 - eureka.client.service-url 353, 359
 - eureka.instance.hostname 352
 - eureka.instance.metadata-map.user.name 474
 - eureka.instance.metadata-map.user. password 474
 - eureka.server.enable-self-preservation 353, 354
 - execution.timeout.enabled 409
 - final 34
 - greeting.message 392
 - href 188
 - Injizieren 124
 - _links 162
 - logging.file 130
 - logging.level 130
 - logging.path 130
 - management.endpoint.health.show-details 427
 - management.endpoints.jmx.exposure.exclude 477
 - management.endpoints.jmx.exposure.include 477
 - management.endpoints.web.exposure.exclude 424
 - management.endpoints.web.exposure.include 424
 - management.endpoint.web.base-path 423
 - Nachrichten 211
 - pageSize 134

- placedAt 79, 132
- propertySources 373
- Protokollierung 130
- security.user.name 126
- security.user.password 126, 136
- server.port 128, 380, 412
- server.ssl.key-store 128
- spring.activemq.broker-url 194
- spring.activemq.in-memory 194
- spring.application.name 358, 378, 461
- spring.boot.admin.client.instance.metadata.user.name 474
- spring.boot.admin.client.instance.metadata.user.password 474
- spring.boot.admin.client.url 460, 461
- spring.cloud.config.discovery.enabled 377
- spring.cloud.config.server.encrypt.enabled 384
- spring.cloud.config.server.git.default-label 375
- spring.cloud.config.server.git.search-paths 375
- spring.cloud.config.server.git.uri 371
- spring.cloud.config.server.password 376
- spring.cloud.config.server.username 376
- spring.cloud.config.token 389
- spring.cloud.config.uri 377
- spring-cloud.version 350
- spring.data.cassandra.contact-points 322
- spring.data.cassandra.keyspace-name 321
- spring.data.cassandra.password 322
- spring.data.cassandra.port 322
- spring.data.cassandra.username 322
- spring.data.mongodb.database 334
- spring.data.mongodb.host 334, 494
- spring.data.mongodb.password 334, 383
- spring.data.mongodb.port 334
- spring.data.mongodb.username 334
- spring.data.rest.base-path 177
- spring.datasource.data 127
- spring.datasource.driver-class-name 127
- spring.datasource.schema 127
- spring.jms.template.default-destination 197
- spring.kafka.bootstrap-servers 218
- spring.kafka.template.default-topic 220
- spring.main.web-application-type 252
- spring.profiles 140
- spring.profiles.active 141, 378
- spring.rabbitmq.template.exchange 210
- spring.rabbitmq.template.receive-timeout 214
- spring.rabbitmq.template.routing-key 210
- systemEnvironment 433
- turbine.app-config 417
- turbine.cluster-name-expression 417
- Werte, spezielle 130
- eingegebettete Datenbanken 333
- Einstiegspunkte 493
- E-Mail, Integrationsflüsse 246
- EmailProperties (Klasse) 247
- EmailToOrderTransformer 249
- EmptyResultDataAccessException 159
- encrypt.key 382
- EndpointRequest 454
- Endpunkte
 - Actuator 423
 - aktivieren/deaktivieren 424
 - Anwendungsaktivität anzeigen 437
 - Anwendungsinformationen abrufen 426
 - /beans 424, 429
 - benutzerdefinierte 177, 451
 - /conditions 424
 - /configprops 429
 - /env 431
 - /health 424
 - /heappdump 437
 - /httptrace 437, 471
 - Hyperlinks hinzufügen 179
 - /info 424
 - Konsumieren 425
 - Laufzeit-Metriken 440
 - /loggers 437
 - /mappings 429, 434
 - /metrics 440
 - /threaddump 438, 470
- Endpunktmodule 245
- End-to-End-Datenflüsse 319
- End-to-End-Stack 290
- Entry Points 493
- equals() 33
- Equals 88
- Ersetzung 156
- Eureka
 - Clienteigenschaften 358
 - Dienstregistrierung 348
 - Dienstregistrierung konfigurieren 352
 - Dienstregistrierung, Überblick 348
 - Lastenausgleich, clientseitiger 349
 - Selbsterhaltungsmodus 353
 - Skalieren 355
- eureka.client.fetch-registry 353

- eureka.client.register-with-eureka 353, 462
- eureka.client.service-url 353, 359
- eureka.instance.hostname 352
- eureka.instance.metadata-map.user.name 474
- eureka.instance.metadata-map.user.password 474
- eureka.server.enable-self-preservation 353, 354
- evenChannel 236
- event looping 286
- exchange() 182, 308
- execute() 79, 182
- executeAndReturnKey() 79
- execution.timeout.enabled 409
- ExecutorChannel 232
- expectStatus() 298
- ExpressionInterceptUrlRegistry 113

- F**
- Fehler
 - behandeln 306
 - HTTP-Statuscode 404 307
 - Nicht gefunden (HTTP 404) 307
 - onStatus() 306
 - Timeout 304
- Fehler und Latenz
 - Fehler überwachen 411
 - Hystrix-Dashboard 412
 - Hystrix-Streams aggregieren 416
 - Hystrix-Threadpools 415
 - Latenz verringern 408
 - Schwellenwerte für Trennschalter 409
 - Trennschalter 403
- Feign (Bibliothek) 363
- FIFO (First In, First Out) 232
- fileWriterChannel 226, 228
- FileWriterGateway 225, 242
- FileWritingMessageHandler 228, 244
- filter() 234, 274
- Filter 233
- Filtern, Daten von reaktiven Typen 272
- final 34
- findAll() 66, 153
 - PagingAndSortingRepository 152
- findById() 66
- findByOrderByCreatedAtDesc() 339
- findByUser() 131
- findByUsername() 106, 312, 331, 341
- findByUserOrderByPlacedAtDesc() 132
- findOne() 67
- First In, First Out (FIFO) 232
- Flapdoodle Embedded MongoDB 333
- flatMap() 276, 308
- Fluent API 163
- Flux, Daten generieren 266
- FluxMessageChannel 232
- follow() 188
- formLogin() 115
- Formulare
 - Eingaben validieren 49
 - Kreditkartennummer 50
 - Validierung bei Formularbindung 52
 - Validierungsfehler anzeigen 53
 - Validierungsregeln 49
 - Verarbeiten 43
- Framework-loses Framework 25
- FreeMarker 57
- from() 243
- fromArray() 265, 318
- fromIterable() 266, 318
- fromStream() 318

- G**
- Gateways 242
- GenericHandler 241
- GenericTransformer 228
- GET-Anfragen
 - Testen 296
 - Verarbeiten 36
- getAuthorities() 106
- getContent() 289
- getForEntity() 184, 185
- getForObject() 184, 185, 302, 360
- getHref() 189
- getImapUrl() 248
- getMessageConverter() 209
- getPrincipal() 121
- Git 370
 - Authentifizieren beim Backend 376
 - Commit-Informationen 445
 - Eigenschaften verschlüsseln 382
 - Konfiguration in Unterordnern 374
- GitHub 370
- GitLab 370
- Gogs 370
 - Benachrichtigungsextraktor 398
- GogsPropertyPathNotificationExtractor 400
- gradle-project 516
- Grails 519
- greeting.message 392
- Groovy Templates 57

- Groß-/Kleinschreibung
 - Ignorieren 89
- groupId 516
- groupSearchBase() 100
- H**
- H2-Konsole 25
- HAL (Hypertext Application Language) 162
- handle() 221, 251
- hashCode() 33
- Hashfunktionen
 - bcrypt 102
 - kryptografische 102
- HashiCorp Vault 385
- hasRole() 113
- HATEOAS (Hypermedia as the Engine of Application State) 160, 425
- helloRouterFunction() 294
- home() 18
- href 188
- HTTP-Anfragen
 - Abbildungen 434
 - Verfolgen 471
- HTTPPie 154, 457
- HTTP-Statuscodes
 - 4xx 306, 307
 - 5xx 306
 - 200 154, 294, 467
 - 201 300
 - 400 467
 - 404 38, 154
 - Metriken abrufen 441
- Hyperlinks, benutzerdefinierte 179
- Hypermedia
 - Aktivieren 160
 - Beziehungen benennen 169
 - Hyperlinks 162
 - Ressourcenassembler 165
- Hypermedia as the Engine of Application State (HATEOAS) 160, 425
- Hystrix
 - Dashboard 412
 - Initializr 406
 - Streams aggregieren 416
 - Threadpools 415
 - Timeout 408
- I**
- IaaS (Infrastructure as a Service) 492
- IgnoresCase 89
- IgnoringCase 89
- Images 492
- imperativer Code 255
- inboundAdapter() 244
- InfoContributor (Schnittstelle) 443
- Informationen anzeigen
 - Controller-Klasse erstellen 34
 - View entwerfen 38
- Infrastructure as a Service (IaaS) 492
- IngredientResource 168
- Initialisieren
 - Anwendungen 6
 - Befehlszeilenoberfläche 517
 - curl und Initializr API 515
 - Projekte mit IntelliJ IDEA 503
 - Projekte mit NetBeans 507
 - Projekte mit Spring Tool Suite 7, 499
 - Projekte unter start.spring.io 511
 - Projekte von der Befehlszeile 515
- Initializr
 - API 515
 - Base Url 8, 502
 - Hystrix 406
- inMemoryAuthentication() 96
- instantiateResource() 167
- IntegrationFlows 230
- Integrationsflüsse
 - Definieren mit XML 225
 - E-Mail 246
- IntelliJ IDEA
 - Projekte initialisieren 503
- Interfaces
 - CrudRepository 87
 - UserDetails 105
 - WebMvcConfigurer 55
- Internet der Dinge 286
- interval() 267
- InventoryService (Bean) 5
- is__Expired() 106
- J**
- Jackson 78
- Jackson2JsonMessageConverter 211
- JAR-Dateien, Verschieben zu Cloud Foundry 489
- Java Cryptography Extensions Unlimited Strength 382
- Java Development Kit 477
- Java, Integrationsflüsse konfigurieren 227
- Java Message Service (JMS) 192
- JavaServer Pages (JSP) 38

Java Streams vs. Reactive Streams 258
 javaVersion 516
 Java virtual machine (JVM) 24
 JConsole 477
 JDBC
 - Arbeiten mit JdbcTemplate 65
 - Benutzerspeicher 97
 - Daten einfügen 72
 - Daten im Voraus laden 69
 - Daten mit JDBC lesen/schreiben 64
 - (Java Database Connectivity) 61
 - Kennwortverschlüsselung 99
 - Repositories definieren 65
 - Schema definieren 69
 - Überschreiben standardmäßiger Benutzerabfragen 97
 JdbcTemplate 26
 - Daten speichern 72
 - SimpleJdbcInsert 76
 - Zeilen einfügen 68
 JDK
 - JConsole 477
 - keytool 128
 JHipster 519
 JMS (Java Message Service) 192
 - Einrichten 192
 - JmsTemplate 194
 - Nachrichten empfangen 202
 - Nachrichtenkonverter konfigurieren 198
 - Nachrichten-Listener deklarieren 204
 - Nachrichten nachbearbeiten 200
 - Nachrichten vor dem Senden konvertieren 198
 JmsOrderMessagingService 195
 jms.send() 196
 JmsTemplate 194
 - Nachrichten empfangen 202
 - Nachrichtenkonverter konfigurieren 198
 - Nachrichten nachbearbeiten 200
 - Nachrichten vor dem Senden konvertieren 198
 JMX
 - Actuator-MBeans 477
 - Benachrichtigungen senden 482
 - (Java Management Extensions) 477
 - MBeans erstellen 480
 jsonPath() 298
 JSP (JavaServer Pages) 38
 just() 261
 JVM (Java virtual machine) 24

K
 Kafka 217
 - Einrichten 218
 - Listener 221
 - Nachrichten mit KafkaTemplate senden 219
 Kanaladapter 243
 Kanäle
 - evenChannel 236
 - Implementierungen 232
 - oddChannel 236
 Kartenprüfnummer 50
 Kennwörter
 - Vergleich konfigurieren 101
 - Verschlüsselung 99
 KeyHolder 74
 keytool 128, 382
 Klassen
 - EmailProperties 247
 - EndpointRequest 454
 - IntegrationFlows 230
 - OrderController 46
 - SimpleJdbcInsert 72
 - WebSecurityConfigurerAdapter 454
 Komponentensuche 5
 - Annotationen 152
 - RegistrationController 109
 Konfiguration
 - and() 115
 - Anwendungsspezifische Eigenschaften 378
 - Beans verknüpfen 124
 - Benutzeroauthentifizierung anpassen 104
 - Benutzerspeicher 95
 - Config Server aktivieren 370
 - Eigenschaften automatisch aktualisieren 393
 - Eigenschaften geheim halten 381
 - Eigenschaften im laufenden Betrieb aktivieren 390
 - Eigenschaften im laufenden Betrieb manuell aktualisieren 391
 - Eigenschaften in Git verschlüsseln 382
 - Eigenschaften injizieren 124
 - Eigenschaften von Profilen bereitstellen 379
 - Feldnamen 117
 - gemeinsame konsumieren 376
 - JDBC-basierter Benutzerspeicher 97
 - Konfigurations-Repository füllen 373
 - LDAP-gestützter Benutzerspeicher 100
 - LDAP-Server 102
 - LDAP-Server, eingebetteter 102
 - Pfad 117

- Pfad der Anmeldeseite 116
- Server, eingebetteter 128
- ServerHttpSecurity 311
- Spring Security 94
- Teilen 368
- Überbrücken 115
- Vault 385
- Verwalten 367
- WebSecurityConfigurerAdapter 95
- Konfigurationseigenschaften 123
 - application.yml 126
 - Autovervollständigung 138
 - Beans bedingt erstellen 142
 - Datenquelle 126
 - Erzeugen eigener 131
 - Holder definieren 134
 - Metadaten 136
 - Optimieren 124
 - Profile 139
 - Profile aktivieren 141
 - profilspezifische 140
 - Protokollierung 129
 - Umgebungsabstraktion 124
 - Werte, spezielle 130
- Kreditkartennummer 50
 - Luhn-Algorithmus 51
- L**
 - language 516
 - ldapAuthentication() 100
 - LDAP (Lightweight Directory Access Protocol)
 - 100
 - Authentifizierung 100
 - Benutzerspeicher 100
 - Kennwortvergleich konfigurieren 101
 - Konfiguration eines eingebetteten LDAP-Servers 102
 - userPassword 101
 - Verweisen auf entfernten LDAP-Server 102
 - LDAP-Server
 - Eingebetteter 102
 - Remote-Server 102
 - url() 102
 - ldif() 103
 - LDIF (LDAP Data Interchange Format) 103
 - Links
 - abfragen 189
 - recents 162
 - self 162
 - LiveReload 24
- loadByUsername() 107
- log() 279
- Logback 129
- loggerLevels 479
- logging.file 130
- logging.level 130
- logging.path 130
- loginPage() 116
- logout() 118
- Lombok 34, 513
- Luhn-Algorithmus 51
- M**
 - Mail imapInboundAdapter() 249
 - main() 15
 - management.endpoint.health.show-details 427
 - management.endpoints.jmx.exposure.exclude 477
 - management.endpoints.jmx.exposure.include 477
 - management.endpoints.web.exposure.exclude 424
 - management.endpoints.web.exposure.include 424
 - management.endpoint.web.base-path 423
 - map() 261
 - MappingJackson2MessageConverter 199
 - mapRowToIngredient() 63
 - Marble-Diagramme 261
 - MarshallingMessageConverter 211
 - matches() 100
 - maven-project 516
 - MBeans
 - Actuator-MBeans 477
 - Erstellen 480
 - MDBs (Message Driven Beans) 191
 - mergeWith() 268
 - Message Driven Beans (MDBs) 191
 - MessagePostProcessor 201, 212
 - MessageProperties 210
 - MessagingMessageConverter 211
 - messsage 51
 - Metadaten
 - Konfigurationseigenschaften 136
 - MeterRegistry 449
 - Methoden
 - Between 88
 - Equals 88
 - Getter- 34
 - Namen 88

- orderForm() 45
 - processDesign() 43, 49
 - processOrder() 49
 - Setter- 34
 - showDesignForm() 37
 - methodOn() 164
 - Michael Nygard 403
 - Micrometer 449
 - Microservices 28, 345
 - Anwendungsserver 489
 - Clienteigenschaften konfigurieren 358
 - Dienste konsumieren 359
 - Dienste registrieren und erkennen 357
 - Dienstregistrierung konfigurieren 352
 - Eureka-Clienteigenschaften 358
 - Eureka-Dienstregistrierung 348
 - Eureka-Dienstregistrierung, Überblick 348
 - Eureka skalieren 355
 - Lastenausgleich, clientseitiger 349
 - Überblick 346
 - Mock-Objekte, TacoRepository 297
 - mongo 428
 - MongoRepository 338
 - MongoTemplate (Bean) 431
 - MPA (Multipage Application) 149
 - Mustache 57
 - MVP *siehe* Minimum Viable Product
- N**
- Nachbearbeiten 200
 - Nachrichten
 - asynchron senden 191
 - Eigenschaften 210
 - Empfangen 202
 - MessagePostProcessor 201
 - Nachbearbeiten 200
 - Polling 215
 - Pull-Modell 202
 - Push-Modell 202
 - Timeout 213
 - Nachrichtenkanäle 232
 - Nachrichtenkonverter
 - MappingJackson2MessageConverter 199
 - SimpleMessageConverter 199
 - name 516
 - NetBeans 507
 - Netflix
 - Ribbon 349
 - Turbine 416
 - NetworkTopologyStrategy 321
- Neustart, automatischer 23
 - ngOnInit() 150
 - NoOpPasswordEncoder 99
 - notes() 452
 - NotesEndpoint 451
 - notes() method 452
 - NotificationPublisherAware (Schnittstelle) 482
 - numberChannel 234
- O**
- Objekte
 - Erzeugen reaktiver Typen 264
 - Objektnachrichten 196
 - oddChannel 236
 - onAfterCreate() 450, 481
 - onComplete() 259
 - onError() 259
 - onNext() 259
 - onStatus() 306
 - onSubmit() 155
 - onSubscribe() 258
 - OpenFeign 363
 - Operationen
 - Ersetzung 156
 - loggerLevels 479
 - OrderBy 132
 - OrderController 46
 - orderForm() 45
 - orderInserter 78
 - ordersForUser() 132
 - OrderSplitter 238
- P**
- packageName 516
 - packaging 516
 - Pageable 132
 - pageSize 134
 - paging 175
 - PagingAndSortingRepository 152, 330
 - Parameter
 - applicationName 516
 - artifactId 516
 - baseDir 516
 - bootVersion 516
 - dependencies 516
 - description 516
 - groupId 516
 - javaVersion 516
 - language 516
 - name 516

- packageName 516
 - packaging 516
 - Pageable 132
 - SessionStatus 80
 - spring init 517
 - type 516
 - version 516
 - ParameterizedTypeReference 188, 215
 - Parsen, Vorlagen 59
 - Partitionsschlüssel 323
 - passwordAttribute() 101
 - passwordCompare() 101
 - passwordEncoder() 99, 101
 - patchOrder() 158
 - pathMatchers() 311
 - PayloadTypeRouter 238
 - Pbkdf2PasswordEncoder 99
 - p-circuit-breaker-dashboard 355
 - p-config-server 355
 - permitAll() 113
 - Persistenz 64
 - Pfade
 - Anmeldeseite 116
 - Platzhalter 153
 - Pivotal Web Services (PWS) 490
 - placedAt 79, 132
 - Platzhalter 153
 - \${} 130
 - Plug-ins 488
 - poChannel 238
 - Polling 215
 - pom.xml, Sicherheit 91
 - Ports 494
 - POST
 - /logout 118
 - Testen 299
 - postForEntity() 186
 - postForLocation() 186
 - postForObject() 186
 - PreparedStatementCreator 74
 - PriorityChannel 232
 - processDesign() 43, 49, 75
 - processOrder() 49
 - processRegistration() 111
 - ProductService (Bean) 5
 - Profile 139
 - Aktivieren 141
 - Beans bedingt erstellen 142
 - cloud 142
 - docker 494
 - Eigenschaften, spezifische 140
 - propertySources 373
 - Protokollierung 130, 435
 - Konfiguration 129
 - p-service-registry 355
 - PublishSubscribeChannel 232
 - Pull 202, 203
 - Operationen 481
 - Push 202
 - Benachrichtigungen 481
 - put() 185
 - PUT 156
 - PWS (Pivotal Web Services) 490
- Q**
- query() 68
 - queryForObject() 63, 67, 68
 - QueueChannel 232
- R**
- RabbitMQ 206
 - Hinzufügen zu Spring 207
 - Nachrichteneigenschaften festlegen 211
 - Nachrichten empfangen 212
 - Nachrichtenkonverter konfigurieren 211
 - Nachrichten mit Listenern verarbeiten 215
 - Nachrichten mit RabbitTemplate empfangen 212
 - Nachrichten senden mit RabbitTemplate 208
 - RabbitTemplate-Beans 207
 - range() 266
 - ReactiveCassandraRepository (Schnittstelle) 329
 - ReactiveCrudRepository 338
 - ReactiveCrudRepository (Schnittstelle) 329
 - ReactiveMongoRepository 338
 - Reactive Streams 257
 - vs. Java Streams 258
 - ReactiveUserDetailsService 312
 - Reactor 255
 - Abhängigkeiten hinzufügen 262
 - BOM (Bill of Materials) 262
 - Datenflüsse grafisch darstellen 261
 - Daten puffern 278
 - Daten von reaktiven Typen filtern 272
 - erste Schritte mit 260
 - Logische Operationen auf reaktiven Typen 281
 - Reaktive Daten abbilden 275
 - Transformieren und filtern 272
 - Typen kombinieren 268

- reaktive APIs 285
 - Anfragen vermitteln 308
 - Controller, reaktive 288
 - Controller testen 296
 - Fehler behandeln 306
 - Funktionale Anfrage-Handler 293
 - Ressourcen löschen 305
 - Ressourcen mit GET abrufen 302
 - Ressourcen senden 304
 - REST APIs reaktiv konsumieren 301
 - Service für Benutzerdetails konfigurieren 311
 - Sichern von Web-APIs 309
 - Spring WebFlux 285
 - Reaktive Programmierung
 - Reactive Streams definieren 257
 - Überblick 256
 - reaktiver Code 255
 - receive() 203, 213
 - receiveAndConvert() 203, 213
 - recents 162
 - Regeln, Vegas- 407
 - registerForm() 109
 - RegistrationController 109
 - Registrieren
 - Admin-Clientanwendungen 460
 - Admin-Clients 460
 - Benutzer 109
 - Release Train 316
 - Config Server 371
 - Eureka-Server 350
 - RendezvousChannel 232
 - Replikation
 - NetworkTopologyStrategy 321
 - SimpleStrategy 321
 - RequestPredicate 294
 - ResourceProcessor 179
 - ResourceSupport 166
 - ResponseStatusException 442
 - Ressourcen
 - Assembler 165
 - löschen 305
 - Pfade anpassen 173
 - RestClientException 407
 - REST-Dienste 147
 - Beziehungen, benennen eingebetteter 169
 - Daten an Server senden 155
 - Daten auf Server aktualisieren 156
 - datengestützte 171
 - Daten vom Server abrufen 150, 154
 - Daten vom Server löschen 159
 - Endpunkte, benutzerdefinierte 177
 - HATEOAS (Hypermedia as the Engine of Application State) 160
 - Hyperlinks 162
 - Hyperlinks zu Spring-Data-Endpunkten hinzufügen 179
 - Hypermedia 160
 - Konsumieren 181
 - Navigieren in REST APIs mit Traverson 187
 - Paging und Sortieren 175
 - RessourcenAssembler 165
 - Ressourcenpfade und Beziehungsnamen anpassen 173
 - REST APIs reaktiv konsumieren 301
 - RESTful Controller 148
 - RestTemplate
 - delete() 186
 - Dienste konsumieren 360
 - getForObject() und getForEntity() 184
 - postForObject() und postForLocation() 186
 - put() 185
 - retrieve() 308
 - Ribbon 349
 - Richtliniendateien 382
 - root() 102
 - route() 237
 - Router 236
 - routerFunction() 295
 - RouterFunction() 294
 - run() 16
 - RXJava, Typen 291
- S**
- save() 68, 72
 - SimpleJdbcInsert-Instanzen 78
 - saveAll() 292, 317
 - saveOrderDetails() 79
 - saveTacoToOrder() 79
 - Scanning 5, 15
 - schema.sql 71, 124
 - Schlüssel
 - Clustering- 323
 - Partitions- 323
 - Schlüsselspeicher 382
 - Schnittstellen
 - AmqpTemplate 208
 - CassandraRepository 330
 - CrudRepository 330, 338
 - InfoContributor 443
 - MongoRepository 338

- NotificationPublisherAware 482
- ReactiveCassandraRepository 329
- ReactiveCrudRepository 329, 338
- ReactiveMongoRepository 338
- scrypt 99
- SCryptPasswordEncoder 99
- SDKMAN 517
- SecurityConfig 95
- SecurityContextHolder 120
- security.user.name 126
- security.user.password 126, 136
- securityWebFilterChain() 311
- Selbsterhaltungsmodus 353
- self 162
- send() 195, 209
- sendDefault() 219
- sendNotification() 482
- SerializerMessageConverter 211
- Server
 - Admin 458, 464
 - Anmelden aktivieren 473
 - Authentifizieren bei Actuator 474
 - contextSource() 102
 - HTTP-Anfragen verfolgen 471
 - Integritätsinformationen 465
 - Konfiguration eines eingebetteten 128
 - Protokollierungsstufen 469
 - Schlüsselmetriken 467
 - Sichern 473
 - Standort konfigurieren 102
 - Threads überwachen 470
 - Umgebungseigenschaften 468
- ServerHttpSecurity 311
- server.port 128, 380, 412
- server.ssl.key-store 128
- SessionStatus 80
- setComplete() 80
- setExpectReply() 228
- setHeader() 212
- setNotificationPublisher() 482
- setStringProperty() 200
- setTypeldMappings() 200
- setTypeldPropertyName() 200
- setViewName() 55
- showDesignForm() 37
- Sicherheit, Spring Security 91
- SimpleJdbcInsert 72
- SimpleJdbcInsert class 72
- SimpleJdbcTemplate 76
- SimpleMessageConverter 199, 211
- Single-Page Application (SPA) 149
- Single Point of Failure 319
- Sitzungen
 - SessionStatus 80
 - Zurücksetzen 80
- skip() 272
- slash() 164
- sort 176
- Sortieren 132, 175
- sources() 488
- SPA (Single-Page Application) 149
- Speicherinterner Benutzerspeicher 96
- SpEL 113
- SpEL (Spring Expression Language) 226
- splitOrderChannel 238
- Splitter 237
- spring.activemq.broker-url 194
- spring.activemq.in-memory 194
- Spring-Anwendungen
 - automatisch neu starten 23
 - Browser-Aktualisierung 24
 - Controller testen 20
 - Erstellen und ausführen 21
 - H2-Konsole 25
 - Initialisieren 6
 - Schreiben 17
 - Spring Boot DevTools 23
 - Testen 16
 - View definieren 19
 - Web-Requests verarbeiten 17, 19
- SpringApplication 16
- spring.application.name 358, 378, 461
- Spring Batch 28
- Spring Boot
 - Autokonfiguration 6
 - Docker-Container 492
 - Überblick 27
- Spring Boot Actuator
 - Anpassen 443
 - Anwendungsaktivität anzeigen 437
 - Anwendungsinformationen abrufen 426
 - Basispfad 423
 - Endpunkte aktivieren/deaktivieren 424
 - Endpunkte, benutzerdefinierte 451
 - Endpunkte konsumieren 425
 - Informationen zum Endpunkt /info beisteuern 443
 - Laufzeit-Metriken 440
 - MBeans 477
 - Metriken, benutzerdefinierte 449

- Sichern 454
- Überblick 421, 423
- Zustandsindikatoren, benutzerdefinierte 448
- Spring Boot Admin 457
 - Clients registrieren 460
 - HTTP-Anfragen verfolgen 471
 - Integritätsinformationen 465
 - Protokollierungsstufen 469
 - Schlüsselmetriken 467
 - Server 464
 - Server erstellen 458
 - Sichern 473
 - Threads überwachen 470
 - Überblick 457
 - Umgebungseigenschaften 468
- spring.boot.admin.client.instance.metadata.
 - user.name 474
- spring.boot.admin.client.instance.metadata.
 - user.password 474
- spring.boot.admin.client.url 460, 461
- Spring Boot DevTools 23
 - Anwendungen automatisch neu starten 23
 - Browser-Aktualisierung 24
 - H2-Konsole 25
 - Template-Cache Deaktivierung 24
- SpringBootServletInitializer 487
- Spring Cloud 28
 - spring.cloud.config.discovery.enabled 377
 - spring.cloud.config.server.encrypt.enabled 384
 - spring.cloud.config.server.git.default-label 375
 - spring.cloud.config.server.git.search-paths 375
 - spring.cloud.config.server.git.uri 371
 - spring.cloud.config.server.password 376
 - spring.cloud.config.server.username 376
 - spring.cloud.config.token 389
 - spring.cloud.config.uri 377
- Spring Cloud Stream 397
- spring-cloud.version 350
- Spring Data 27, 315
 - Cassandra aktivieren 320
 - Cassandra-Datenmodellierung 323
 - Domänenotypen auf Dokumente abbilden 334
 - Domänenotypen für Cassandra-Persistenz 323
 - Konvertieren zwischen reaktiven und nichtreaktiven Typen 317
 - Methodensignaturen 88
 - MongoDB aktivieren 332
 - reaktive Cassandra-Repositories 319, 329
 - reaktive MongoDB-Repository-Schnittstellen 338
 - reaktive Repositories entwickeln 319
 - reaktive Repositories mit MongoDB 332
 - spring.data.cassandra.contact-points 322
 - spring.data.cassandra.keyspace-name 321
 - spring.data.cassandra.password 322
 - spring.data.cassandra.port 322
 - spring.data.cassandra.username 322
- Spring Data JPA (Java Persistence API) 81
 - Domäne als Entitäten annotieren 82
 - Hinzufügen zum Projekt 81
 - JPA-Repositories anpassen 87
 - JPA-Repositories deklarieren 86
- Spring Data MongoDB 332
 - Aktivieren 332
 - reaktive Repository-Schnittstellen 338
- spring.data.mongodb.database 334
- spring.data.mongodb.host 334, 494
- spring.data.mongodb.password 334, 383
- spring.data.mongodb.port 334
- spring.data.mongodb.username 334
- Spring Data Reactive Cassandra 321
- spring.data.rest.base-path 177
- spring.datasource.data 127
- spring.datasource.driver-class-name 127
- spring.datasource.schema 127
- Spring Expression Language (SpEL) 226
- Spring Framework 26
- spring init 517
 - Parameter 517
- Spring Integration 28, 223
 - Dienstaktivatoren 240
 - DSL-Konfiguration 229
 - E-Mail-Integrationsfluss 246
 - Endpunktmodule 245
 - Filter 233
 - Gateways 242
 - Integrationsflüsse 224
 - Integrationsflüsse in Java konfigurieren 227
 - Kanaladapter 243
 - Kanalimplementierungen 232
 - Nachrichtenkanäle 232
 - Router 236
 - Splitter 237
 - Transformer 234
- spring.jms.template.default-destination 197
- SpringJUnit4ClassRunner 17
- spring.kafka.bootstrap-servers 218
- spring.kafka.template.default-topic 220
- spring.main.web-application-type 252
- spring.profiles 140

- spring.profiles.active 141, 378
- SPRING_PROFILES_ACTIVE 378, 494
- Spring-Projekte
 - Anwendungen testen 16
 - Anwendung hochfahren 15
 - Build-Spezifikation 12
 - Initialisieren mit IntelliJ IDEA 503
 - Initialisieren mit NetBeans 507
 - Initialisieren mit Spring Tool Suite 7, 499
 - Initialisieren unter start.spring.io 511
 - Initialisieren von der Befehlszeile 515
 - SpringRunner 17
 - Struktur 11
- spring.rabbitmq.template.exchange 210
- spring.rabbitmq.template.receive-timeout 214
- spring.rabbitmq.template.routing-key 210
- SpringRunner 17
- Spring Security 28, 91
 - Abmelden 118
 - Aktivieren 91
 - Anfragen sichern 112
 - Anmeldeseite erstellen 115
 - Benutzerauthentifizierung anpassen 104
 - Benutzer ermitteln 120
 - Benutzerspeicher, speicherinterner 96
 - CSRF-Angriffe 118
 - JDBC-basierter Benutzerspeicher 97
 - Konfigurieren 94
 - LDAP-gestützter Benutzerspeicher 100
 - Webanfragen sichern 112
- Spring Tool Suite
 - Projekte initialisieren 499
 - Spring-Projekte initialisieren mit 7
- Spring, Überblick 6
- Spring WebFlux 285
 - Controller, reaktive 288
 - Eingaben reaktiv verarbeiten 292
 - Einzelwerte zurückgeben 290
 - RXJava-Typen 291
 - Starterabhängigkeit 287
 - Überblick 287
- SQLException 63
- StandardPasswordEncoder 99
- start.spring.io, Projekte initialisieren 511
- StepVerifier 264
- Streams, Backpressure 257
- StreamUtils 298
- String 89
- subscribe() 258
- System.currentTimeMillis() 219
- systemEnvironment 433
- T
- taco.orders.pageSize 133
- TacoRepository
 - findAll() 153
 - Mock-Objekt 297
- tag 442
- Tags
 - tag 442
 - uri 442
- take() 267, 273
- tar 517
- Template-Cache, Deaktivierung 24
- Template-Engine 18
- Testen
 - Anwendungen 16
 - Controller 20
 - Controller, reaktive 296
 - GET-Anfragen 296
 - Live-Server 300
 - POST-Anfragen 299
- testHomePage() 21
- testTaco() 297
- textInChannel 228
- th:each 39
- th:errors 53
- Threads, Überwachen 470
- th:text 39
- Thymeleaf 14, 18, 513
 - Attribute 39
 - Errors-Objekt 53
- timeout() 304
- Timeout
 - Fehlerbehandlung 304
 - Nachrichtenempfang 213
 - subscribe() 304
- to() 455
- toAnyEndpoint() 455
- tolerable() 318
- toObject() 188
- Tools
 - keytool 382
 - Schlüssel erzeugen 382
 - vault 385
- toResource() 167
- toRoman() 235
- toString() 33
- toUser() 111
- Transformer 234

- EmailToOrderTransformer 249
- transformerFlow() 235
- Traverson, Navigieren in REST APIs 187
- Trennschalter 403
 - deklarieren 405
 - Latenz reduzieren 408
 - Schwellenwerte 409
- Tuple2 270
- Turbine 416
- turbine.app-config 417
- turbine.cluster-name-expression 417
- type 516
- Typen
 - abstrakte 326
 - benutzerdefinierte 326
 - UDT (User-Defined Type) 326
- U**
 - Überschreiben, standardmäßige
 - Benutzerabfragen 97
 - Überwachen, Threads 470
 - UDT (User-Defined Type) 326
 - UI (User Interface) *siehe* Benutzeroberflächen
 - Umgebung, Abstraktion 124
 - Umgebungseigenschaften, Admin-Server 468
 - Umgebungsvariablen
 - Profile aktivieren 141
 - SPRING_PROFILES_ACTIVE 378, 494
 - taco.orders.pageSize 133
 - VCAP_SERVICES 491
 - update() 68, 72
 - uppercase() 242
 - uri 442
 - url() 102
 - UserDetails 105
 - userDetailsService() 108, 311
 - User Interface (UI) *siehe* Benutzeroberflächen
 - UsernameNotFoundException 106
 - userPassword 101
 - UserRepositoryUserDetailsService 107
 - userSearchBase() 100
 - userSearchFilter() 100
 - users.Idif 103
- V**
 - Validierung
 - Fehler anzeigen 53
 - Formularbindung 52
 - Formulareingaben 49
 - Regeln deklarieren 49
- Vault 385
 - Anwendungs- und profilspezifische Geheimnisse schreiben 389
 - Backend in Config Server aktivieren 387
 - Geheimnisse schreiben 386
 - Server starten 385
 - Token in Config Server-Clients setzen 389
- VCAP_SERVICES 491
- Vegas-Regel 407
- Verfolgen, HTTP-Anfragen 471
- Verschlüsselung
 - Datenbanken 99
 - Kennwörter 99
 - passwordEncoder() 99, 101
- version 516
- Views
 - Controller 55
 - Entwerfen 38
 - Templates 57, 59
- Vorlagen, Parsen 59
- W**
 - WAR-Dateien 487
 - Web-Abhängigkeit 513
 - Webanfragen
 - Abmelden 118
 - Anmeldeseite erstellen 115
 - CSRF-Angriffe 118
 - Sichern 112
 - Webanwendungen
 - Controller-Klasse erstellen 34
 - Domäne einrichten 33
 - Entwickeln 31
 - Formulareingaben validieren 49
 - Formulare verarbeiten 43
 - Informationen anzeigen 31
 - Validierung bei Formularbindung 52
 - Validierungsfehler anzeigen 53
 - Validierungsregeln 49
 - View-Controller 55
 - View entwerfen 38
 - View-Template-Bibliothek auswählen 57
 - WebApplicationInitializer 487
 - WebClient
 - Dienste konsumieren 361
 - HTTP-Statuscodes 306
 - WebClientResponseException 306
 - webEnvironment 300
 - Webhooks
 - Erstellen 394

- Überblick 393
 - Updates in Config Server behandeln 397
 - WebMvcConfigurer 55
 - WebSecurityConfigurerAdapter 95, 310, 454
 - configure() 112
 - WebTestClient 298
 - where-Klausel 331
 - withDetail() 444
 - withUser() 96
 - Wrapper-Skripts, Maven 11
 - writeToFile() 225
- X**
- X-Config-Token 388
 - X_ORDER_SOURCE 201
 - X-Vault-Token 388
- Z**
- zip() 269