KAPITEL 4

Validierung und Formatierung

Dieses Kapitel enthält Rezepte für das Validieren und Formatieren von Benutzereingaben. Manche der Lösungen zeigen, wie man gültige Eingaben überprüft, so zum Beispiel Postleitzahlen in den USA, die entweder fünf oder neun Ziffern enthalten können. Andere sind dazu da, allgemein anerkannte Formatierungsrichtlinien auf Daten anzuwenden, wie zum Beispiel bei Telefonnummern, Datumswerten und Kreditkartennummern.

Abgesehen davon, dass Regexes Ihnen dabei helfen, ungültige Eingaben zu verhindern, können diese Rezepte auch die Benutzerfreundlichkeit Ihrer Anwendungen verbessern. Meldungen wie „keine Leerzeichen oder Bindestriche“ neben Feldern für Kreditkartennummern frustrieren die Leute häufig oder werden einfach ignoriert. Reguläre Ausdrücke sorgen dafür, dass die Anwender die Daten so eingeben können, wie es für sie am einfachsten ist, ohne dass Sie hinterher zu viel Arbeit damit haben.

Einige Programmiersprachen stellen über eingebaute Klassen oder Bibliotheken Funktionalitäten bereit, die der mancher Rezepte ähneln. Je nachdem, was Sie brauchen, kann es sinn-voller sein, auf diese Optionen zurückzugreifen, daher werden wir sie – wo es passt – erwähnen.

4.1 E-Mail-Adressen überprüfen

Problem

Sie haben ein Formular auf Ihrer Website oder ein Eingabefeld in Ihrer Anwendung, in das der Benutzer eine E-Mail-Adresse eingeben soll. Sie wollen einen regulären Ausdruck verwenden, um diese E-Mail-Adresse zu validieren, bevor Sie versuchen, eine E-Mail dorthin zu schicken. Damit wird die Menge an E-Mails reduziert, die als unzustellbar an Sie zurückgehen.

Lösung

Einfach

Die einfache Lösung führt eine sehr simple Validierung durch. Sie prüft nur, ob die E-Mail-Adresse ein einzelnes At-Zeichen (@) besitzt und keinen Whitespace enthält:

```
^\S+@\S+$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A\S+@\S+\Z
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Einfach, mit Einschränkung der Zeichen

Der *Domainname*, also der Teil nach dem @-Zeichen, ist auf bestimmte Zeichen beschränkt. Der *Benutzername*, der Teil vor dem @-Zeichen, ist das ebenfalls, wobei die meisten E-Mail-Clients und -Server bei der Akzeptanz etwas großzügiger verfahren:

```
^[A-Z0-9+_.-]+@[A-Z0-9.-]+$
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[A-Z0-9+_.-]+@[A-Z0-9.-]+\Z
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Einfach, mit allen Zeichen

Dieser reguläre Ausdruck erweitert den vorigen, indem er mehr (selten genutzte) Zeichen im Benutzernamen zulässt. Nicht jede E-Mail-Software kann mit all diesen Zeichen umgehen, aber wir haben alle Zeichen aufgenommen, die nach dem RFC 2822 zulässig sind. In diesem RFC ist das E-Mail-Format definiert. Unter diesen Zeichen gibt es ein paar, die durchaus ein Sicherheitsrisiko darstellen können, wenn sie als Benutzereingabe direkt an eine SQL-Anweisung übergeben werden – so zum Beispiel das einfache Anführungszeichen (') und der vertikale Strich (|). Stellen Sie immer sicher, dass kritische Zeichen maskiert sind, wenn Sie die E-Mail-Adresse in einen String für ein anderes Programm einfügen. Damit vermeiden Sie Sicherheitslecks, zum Beispiel durch SQL-Injection-Angriffe:

```
^[\w!#$%&'*+/=?`{|}~^.-]+@[A-Z0-9.-]+$
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[\w!#$%&'*+/=?`{|}~^.-]+@[A-Z0-9.-]+\Z
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Keine Punkte am Anfang, am Ende und direkt nacheinander

Sowohl der Benutzername als auch der Domainname dürfen mehr als einen Punkt enthalten, aber es dürfen keine zwei Punkte direkt aufeinanderfolgen. Zudem dürfen weder das erste noch das letzte Zeichen im Benutzer- und Domainnamen Punkte sein:

```
^[\w!#$%&'*+/=?`{|}~^-]+(?:\.[\w!#$%&'*+/=?`{|}~^-]+)*@: "
[A-Z0-9-]+(?:\.[A-Z0-9-]+)*$
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[\w!#$%&'*+/=?`{|}~^-]+(?:\.[\w!#$%&'*+/=?`{|}~^-]+)*@: "
[A-Z0-9-]+(?:\.[A-Z0-9-]+)*\Z
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Top-Level-Domain besitzt zwischen zwei und sechs Zeichen

Dieser reguläre Ausdruck ergänzt die vorherigen Versionen um weitere Regeln für den Domainnamen: Er muss mindestens ein Zeichen enthalten, und der Teil des Domainnamens nach dem Punkt darf nur aus Buchstaben bestehen. Somit muss die Domain mindestens zwei Level besitzen, wie zum Beispiel `secondlevel.de` oder `thirdlevel.secondlevel.de`. Die Top-Level-Domain, hier `.de`, muss zwischen zwei und sechs Buchstaben enthalten. Alle länderspezifischen Top-Level-Domains haben nur zwei Buchstaben. Die generischen Top-Level-Domains besitzen zwischen drei (`.com`) und sechs Zeichen (`.museum`):

```
^[\w!#$%&'*+/=?`{|}~^-]+(?:\.[\w!#$%&'*+/=?`{|}~^-]+)*@: "
(?:[A-Z0-9-]+\.)+[A-Z]{2,6}$
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[\w!#$%&'*+/=?`{|}~^-]+(?:\.[\w!#$%&'*+/=?`{|}~^-]+)*@: "
(?:[A-Z0-9-]+\.)+[A-Z]{2,6}\Z
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Diskussion

E-Mail-Adressen

Wenn Sie dachten, für etwas konzeptionell so Einfaches wie das Validieren einer E-Mail-Adresse gäbe es eine schlichte Regex-Lösung, die allen Erfordernissen gerecht wird, lie-

gen Sie leider ziemlich falsch. Dieses Rezept ist ein wundervolles Beispiel für die Tatsache, dass Sie vor dem Schreiben eines regulären Ausdrucks *genau* wissen müssen, was Sie finden wollen. Es gibt keine allgemein anerkannte Regel dafür, welche E-Mail-Adressen gültig sind und welche nicht. Das hängt nämlich stark von Ihrer Definition von *gültig* ab.

`asdf@asdf.asdf` ist laut RFC 2822 (der Syntaxdefinition für E-Mail-Adressen) gültig. Aber sie ist nicht gültig, wenn Sie festlegen, dass eine valide E-Mail-Adresse auch Mails annehmen muss. Denn es gibt keine Top-Level-Domain `asdf`.

Die kurze Antwort auf das Gültigkeitsproblem ist, dass Sie nicht wissen können, ob `monika.mustermann@irgendwo.de` eine E-Mail-Adresse ist, die tatsächlich E-Mails empfangen kann, bis Sie versuchen, dorthin eine E-Mail zu schicken. Und selbst dann können Sie nicht sicher sein, ob eine fehlende Reaktion auf die E-Mail zeigt, dass die Domain `irgendwo.de` still und leise Mails verwirft, die an nicht vorhandene Mailboxen geschickt werden, oder dass Monika Mustermann auf den Lösch-Button geklickt hat beziehungsweise der Spam-Filter diese Aufgabe übernommen hat.

Da Sie letztendlich immer eine E-Mail verschicken müssen, wenn Sie sicherstellen wollen, dass die entsprechende Adresse existiert, können Sie sich auch dazu entschließen, einen einfacheren, etwas laxeren regulären Ausdruck zu nutzen. Es kann durchaus sinnvoll sein, die eine oder andere ungültige E-Mail-Adresse durchrutschen zu lassen, statt die Leute damit zu verärgern, ihre gültigen E-Mail-Adressen abzulehnen. Aus diesem Grund verwenden Sie ruhig den regulären Ausdruck für „Einfach, mit allen Zeichen“. Obwohl er ganz offensichtlich viele Einträge erlaubt, die keine E-Mail-Adressen sind, wie zum Beispiel `#$%@.-`, ist die Regex schnell und einfach, zudem wird sie niemals gültige E-Mail-Adressen ungerechtfertigterweise ablehnen.

Wenn Sie vermeiden wollen, zu viele nicht zustellbare E-Mails zu verschicken, aber trotzdem keine echten E-Mail-Adressen ablehnen wollen, ist die Regex in „Top-Level-Domain besitzt zwischen zwei und sechs Zeichen“ auf Seite 229 eine gute Wahl.

Sie müssen sich überlegen, wie umfassend Ihr regulärer Ausdruck sein soll. Wenn Sie Benutzereingaben validieren, werden Sie vermutlich eine eher komplexe Regex nutzen, da der Anwender alles Mögliche eingeben kann. Aber wenn Sie Datenbankdateien durchsuchen, bei denen Sie wissen, dass sie nur gültige E-Mail-Adressen enthalten, können Sie eine sehr einfache Regex nutzen, die eigentlich nur die E-Mail-Adressen von den anderen Daten separiert. Selbst die Lösung im Abschnitt „Einfach“ kann dann ausreichen.

Schließlich müssen Sie sich auch noch Gedanken darüber machen, wie zukunftssicher Ihr regulärer Ausdruck sein soll. In der Vergangenheit war es sinnvoll, die Top-Level-Domain auf Zwei-Buchstaben-Kombinationen für die Länderkürzel zu beschränken und dazu eine vollständige Liste der generischen Top-Level-Domains anzugeben, also `‹com|net|org|mil|edu›`. Dadurch dass jetzt immer wieder neue Top-Level-Domains hinzukommen, veralten solche regulären Ausdrücke aber recht schnell.

Regex-Syntax

Die in diesem Rezept vorgestellten regulären Ausdrücke enthalten bereits alle Grundelemente der Regex-Syntax. Wenn Sie sich dazu Kapitel 2 durchlesen, können Sie schon 90% aller Aufgaben erledigen, die sich durch reguläre Ausdrücke am besten lösen lassen.

Bei allen hier genutzten regulären Ausdrücken muss die Option zum Ignorieren von Groß- und Kleinschreibung eingeschaltet sein. Ansonsten wären nur Großbuchstaben zulässig. Schaltet man die Option ein, können Sie ‹[A-Z]› statt ‹[A-Za-z]› eingeben und damit ein paar Tastendrücke sparen. Wenn Sie einen der letzten beiden regulären Ausdrücke verwenden, ist diese Option sehr praktisch. Ansonsten müssten Sie jedes Zeichen ‹X› durch ‹[Xx]› ersetzen.

‹\S› und ‹\w› sind Zeichenklassen, die im Rezept in Rezept 2.3 beschrieben wurden. ‹\S› passt zu jedem Nicht-Whitespace-Zeichen, während ‹\w› ein Wortzeichen findet.

‹@› und ‹\.› passen zu einem literalen @-Zeichen beziehungsweise zu einem Punkt. Da der Punkt ein Metazeichen ist, wenn man ihn außerhalb von Zeichenklassen verwendet, muss er mit einem Backslash maskiert werden. Das @-Zeichen hat in keiner der in diesem Buch behandelten Regex-Varianten eine besondere Bedeutung. In Rezept 2.1 finden Sie eine Liste aller Metazeichen, die maskiert werden müssen.

‹[A-Z0-9.-]› und die anderen Sequenzen zwischen eckigen Klammern sind Zeichenklassen. Diese lässt alle Zeichen von A bis Z, alle Ziffern von 0 bis 9 sowie einen literalen Punkt und einen Bindestrich zu. Auch wenn der Bindestrich in einer Zeichenklasse normalerweise für den Aufbau einer Sequenz genutzt wird, wird er als literaler Strich behandelt, wenn er das letzte Zeichen der Zeichenklasse ist. Das Rezept 2.3 erklärt alles Notwendige über Zeichenklassen, einschließlich der Kombinationen mit Abkürzungen, wie in ‹[\w!#$%&'*+/=?`{|}~^.-]›. Diese Klasse passt zu einem Wortzeichen, aber auch zu einem beliebigen anderen der 19 aufgeführten Satzzeichen.

‹+› und ‹*› sind, wenn man sie außerhalb von Zeichenklassen verwendet, Quantoren. Das Pluszeichen wiederholt das vorige Regex-Token ein Mal oder mehrmals, während der Stern das Token null Mal oder mehrmals wiederholt. In diesen regulären Ausdrücken ist das zu wiederholende Token normalerweise eine Zeichenklasse, manchmal auch eine Gruppe. Daher passt ‹[A-Z0-9.-]+› zu einem oder mehreren Buchstaben, Ziffern, Punkten und/oder Bindestrichen.

Als Beispiel für die Verwendung einer Gruppe passt ‹(?:[A-Z0-9-]+\.)+› zu einem oder mehreren Zeichen, Ziffern und/oder Bindestrichen, gefolgt von einem literalen Punkt. Das Pluszeichen wiederholt diese Gruppe ein Mal oder mehrmals. Die Gruppe muss mindestens einmal passen, kann aber auch beliebig häufig vorkommen. In Rezept 2.12 werden die Mechanismen solcher Konstrukte im Detail beschrieben.

‹(?:Gruppe)› ist eine nicht-einfangende Gruppe. Verwenden Sie sie, um eine Gruppe aus einem Teil des regulären Ausdrucks zu erstellen, sodass Sie einen Quantor auf die ganze Gruppe anwenden können. Die einfangende Gruppe ‹(Gruppe)› sorgt für das Gleiche mit einer einfacheren Syntax, daher können Sie in allen bisher genutzten regulären Ausdrücken ‹(?:› durch ‹(› ersetzen, ohne die Suchergebnisse zu verändern.

Aber da wir nicht daran interessiert sind, Teile der E-Mail-Adresse getrennt einzufangen, ist die nicht-einfangende Gruppe effizienter, auch wenn der reguläre Ausdruck dadurch schlechter lesbar wird. In Rezept 2.9 erfahren Sie alles über einfangende und nicht-einfangende Gruppen.

Die Anker ‹^› und ‹$› zwingen den regulären Ausdruck dazu, die Übereinstimmung am Anfang beziehungsweise Ende des Ausgangstexts zu finden. Setzt man den gesamten regulären Ausdruck zwischen diese Zeichen, wird damit dafür gesorgt, dass der gesamte Text passen muss.

Das ist wichtig, wenn man Benutzereingaben validiert, denn Sie wollen ja `drop database; -- joe@server.com Haha!` als gültige E-Mail-Adresse nicht wirklich akzeptieren. Ohne die Anker würden alle aufgeführten regulären Ausdrücke passen, weil sie `joe@server.com` in der Mitte des angegebenen Texts finden. In Rezept 2.5 wird die Sache detaillierter beschrieben. Dieses Rezept erklärt auch, warum die Option *Zirkumflex und Dollar passen auf Zeilenumbrüche* abgeschaltet sein muss.

In Ruby passen Zirkumflex und Dollarzeichen immer auf Zeilenumbrüche. Der reguläre Ausdruck mit Zirkumflex und Dollarzeichen funktioniert in Ruby korrekt, aber nur wenn der String, den Sie validieren wollen, keine Zeilenumbrüche enthält. Enthält der String eventuell Zeilenumbrüche, passen alle Regexes mit ‹^› und ‹$› auf die E-Mail-Adresse in `drop database; --` LF `joe@server.com` LF `Haha!`, wobei LF für einen Zeilenumbruch steht.

Um das zu vermeiden, sollten Sie stattdessen die Anker ‹\A› und ‹\Z› verwenden. Diese passen nur am Anfang und am Ende des Strings – egal mit welcher Option und in welcher Regex-Variante, jedoch mit der Ausnahme von JavaScript. JavaScript kennt ‹\A› und ‹\Z› gar nicht. In Rezept 2.5 werden diese Anker erklärt.

Der Unterschied zwischen ‹^› und ‹$› sowie ‹\A› und ‹\Z› betrifft alle regulären Ausdrücke, die Benutzereingaben überprüfen. Es gibt eine Reihe von diesen Regexes in diesem Buch. Wir werden zwar gelegentlich an den Unterschied erinnern, aber nicht ständig diesen Rat wiederholen oder getrennte Lösungen für JavaScript und Ruby anbieten. In vielen Fällen werden wir nur eine Lösung mit Zirkumflex und Dollar vorstellen und Ruby als kompatible Variante aufführen. Nutzen Sie Ruby, sollten Sie daran denken, ‹\A› und ‹\Z› zu verwenden, wenn Sie das Problem mit den einzelnen Zeilen in einem mehrzeiligen String vermeiden wollen.

Eine Regex Schritt für Schritt aufbauen

Dieses Rezept zeigt Ihnen, wie Sie einen regulären Ausdruck Schritt für Schritt aufbauen können. Diese Technik ist insbesondere bei einem interaktiven Regex-Tester nützlich, wie zum Beispiel bei RegexBuddy.

Zunächst laden Sie eine Reihe gültiger und ungültiger Beispieldaten in das Tool. In diesem Fall wäre das eine Liste gültiger und ungültiger E-Mail-Adressen.

Dann schreiben Sie einen einfachen regulären Ausdruck, der alle gültigen E-Mail-Adressen findet. Ignorieren Sie zunächst die ungültigen Adressen. ‹^\S+@\S+$› zeigt schon die grundlegende Struktur einer E-Mail-Adresse: einen Benutzernamen, ein At-Zeichen und einen Domainnamen.

Haben Sie die prinzipielle Struktur Ihres Textmusters definiert, können Sie jeden Teil so lange verbessern, bis der reguläre Ausdruck keine ungültigen Daten mehr findet. Soll die Regex nur mit schon vorhandenen Daten arbeiten, kann das schnell gehen. Muss aber eine beliebige Benutzereingabe verarbeitet werden, ist es viel schwerer, ihren regulären Ausdruck so restriktiv zu machen, dass er wirklich nur gültige Daten erfasst.

Variationen

Wenn Sie in größeren Textmengen nach E-Mail-Adressen suchen wollen, können Sie die Anker ‹^› und ‹$› nicht verwenden. Sie einfach nur aus der Regex zu entfernen, ist aber auch nicht korrekt. Bei der abschließenden Regex, die die Top-Level-Domain auf Buchstaben beschränkt, würden Sie so zum Beispiel auch `asdf@asdf.as` in `asdf@asdf.as99` finden. Statt die Regex-Übereinstimmung am Anfang und Ende des Gesamttexts zu verankern, müssen Sie stattdessen festlegen, dass der Anfang des Benutzernamens und die Top-Level-Domain kein Teil längerer Wörter sein darf.

Das lässt sich leicht mit einem Paar Wortgrenzen erreichen. Ersetzen Sie ‹^› und ‹$› durch ‹\b›. So wird zum Beispiel ‹^[A-Z0-9+_.-]+@(?:[A-Z0-9-]+\.)+[A-Z]{2,6}$› zu ‹\b[A-Z0-9+_.-]+@(?:[A-Z0-9-]+\.)+[A-Z]{2,6}\b›.

Diese Regex kombiniert den Benutzernamen-Teil aus „Einfach, mit Einschränkung der Zeichen“ auf Seite 228 mit dem Domainnamen-Teil aus „Top-Level-Domain besitzt zwischen zwei und sechs Zeichen“ auf Seite 229. Wir haben festgestellt, dass dieser reguläre Ausdruck in der Praxis ziemlich gut funktioniert.

Siehe auch

Der RFC 2822 legt Struktur und Syntax von E-Mail-Nachrichten fest – einschließlich der E-Mail-Adressen, die in solchen Nachrichten verwendet werden. Sie können ihn unter *http://www.ietf.org/rfc/rfc2822.txt* herunterladen.

4.2 Nordamerikanische Telefonnummern validieren

Problem

Sie wollen herausfinden, ob ein Benutzer eine Telefonnummer aus Nordamerika in einem gebräuchlichen Format eingegeben hat – einschließlich der Vorwahl. Zu diesen Formaten gehören `1234567890`, `123-456-7890`, `123.456.7890`, `123 456 7890`, `(123) 456 7890` und alle entsprechenden Kombinationen. Ist die Telefonnummer gültig, wollen Sie sie in Ihr Standardformat `(123) 456-7890` umwandeln, sodass Ihre Telefonnummerndaten konsistent sind.

Lösung

Ein regulärer Ausdruck kann leicht herausfinden, ob ein Benutzer etwas eingegeben hat, das wie eine gültige Telefonnummer aussieht. Durch einfangende Gruppen, die sich jeden Ziffernbereich merken, kann der gleiche reguläre Ausdruck auch genutzt werden, um den Ausgangstext durch das Format zu ersetzen, das Sie haben wollen.

Regulärer Ausdruck

```
^\(?([0-9]{3})\)?[-.”]?([0-9]{3})[-.”]?([0-9]{4})$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ersetzungstext

```
($1)”$2-$3
```

Ersetzungstextvarianten: .NET, Java, JavaScript, Perl, PHP

```
(\1)”\2-\3
```

Ersetzungstextvarianten: Python, Ruby

C#

```
Regex regexObj =
    new Regex(@"^\(?([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$");

if (regexObj.IsMatch(subjectString)) {
    string formattedPhoneNumber =
        regexObj.Replace(subjectString, "($1) $2-$3");
} else {
    // Ungültige Telefonnummer
}
```

JavaScript

```
var regexObj = /^\(?([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$/;
if (regexObj.test(subjectString)) {
    var formattedPhoneNumber =
        subjectString.replace(regexObj, "($1) $2-$3");
} else {
    // Ungültige Telefonnummer
}
```

Andere Programmiersprachen

In den Rezepten 3.5 und 3.15 finden Sie Beispiele für das Implementieren dieses regulären Ausdrucks mit anderen Programmiersprachen.

Diskussion

Dieser reguläre Ausdruck passt auf drei Zifferngruppen. Die erste Gruppe kann optional von Klammern umschlossen sein, und auf die ersten beiden Gruppen kann optional eines von drei Trennzeichen folgen (ein Bindestrich, ein Punkt oder ein Leerzeichen). Das folgende Layout unterteilt den regulären Ausdruck in seine einzelnen Teile, wobei die redundanten Zifferngruppen weggelassen werden:

```
^        # Übereinstimmung am String-Anfang sicherstellen.
\(       # Literale "(" ...
  ?      #   null oder ein Mal finden.
(        # Folgende Übereinstimmung in Rückwärtsreferenz 1 einfangen ...
  [0-9]  #   Eine Ziffer ...
    {3}  #     genau drei Mal finden.
)        # Ende der einfangenden Gruppe 1.
\)       # Literale ")" ...
  ?      #   null- oder einmal finden.
[-. ]    # Eines der Zeichen aus "-. " ...
  ?      #   null oder ein Mal finden.
...      # [Restliche Ziffern und Trennzeichen finden.]
$        # Übereinstimmung am String-Ende sicherstellen.
```

Lassen Sie uns jeden dieser Teile genauer anschauen.

Die Zeichen ‹^› und ‹$› am Anfang und Ende des regulären Ausdrucks sind besondere Metazeichen, sogenannte *Anker* oder *Assertions*. Sie finden keinen Text, sondern eine Position im Text. So stellt ‹^› sicher, dass die Übereinstimmung am Anfang des Texts liegen muss, während ‹$› nur am Ende des Texts passt. Damit haben Sie veranlasst, dass die Telefonnummern-Regex zu keinem längeren Text passt, wie zum Beispiel `123-456-78901`.

Wie wir wiederholt gesehen haben, sind Klammern in regulären Ausdrücken besondere Zeichen, aber in diesem Fall wollen wir es dem Anwender ermöglichen, Klammern einzugeben und sie auch von unserer Regex erkennen zu lassen. Dies ist ein Paradebeispiel für die Verwendung eines Backslashs, um ein Sonderzeichen so zu maskieren, dass der reguläre Ausdruck es als literale Eingabe behandelt. Daher passen die Sequenzen ‹\(› und ‹\)›, die die erste Zifferngruppe umschließen, auf literale Klammerzeichen. Beiden folgt ein Fragezeichen, was bedeutet, dass sie optional sind. Wir werden das Fragezeichen noch genauer behandeln, wollen aber erst die anderen Arten von Tokens in diesem regulären Ausdruck besprechen.

Die Klammern, die ohne Backslashs genutzt werden, sind einfangende Gruppen, die später verwendet werden, um sich die daHigefundenen Werte zu merken und sie wieder aufzurufen. In diesem Fall werden Rückwärtsreferenzen auf die eingefangenen Werte im Ersetzungstext angewendet, damit wir die Telefonnummer wie gewünscht umformatieren können.

Zwei weitere Arten von Tokens, die in diesem regulären Ausdruck verwendet werden, sind Zeichenklassen und Quantoren. Zeichenklassen ermöglichen es Ihnen, ein beliebiges Zeichen aus einer Menge von Zeichen zu finden. ‹[0-9]› ist eine Zeichenklasse, die

eine beliebige Ziffer findet. Die in diesem Buch behandelten Regex-Varianten bieten alle die Zeichenklassenabkürzung ‹\d› an, die ebenfalls eine Ziffer findet, allerdings werden in manchen Varianten damit auch Ziffern aus beliebigen Schriftsystemen oder Sprachen gefunden. Das wollen wir hier natürlich nicht. In Rezept 2.3 erhalten Sie mehr Informationen über ‹\d›.

‹[-."]› ist eine weitere Zeichenklasse, durch die eines von drei möglichen Trennzeichen gefunden werden kann. Es ist wichtig, dass der Bindestrich als Erstes in der Zeichenklasse vorkommt, denn stünde er zwischen anderen Zeichen, würde er wie in ‹[0-9]› zu einem Bereich führen. Man kann den Bindestrich auch in einer Zeichenklasse mit einem Backslash maskieren. ‹[.\-"]› findet daher die gleichen Zeichen.

Quantoren schließlich ermöglichen es Ihnen, ein Token oder eine Gruppe zu wiederholen. ‹{3}› ist ein Quantor, der dafür sorgt, dass das vorherige Element genau drei Mal wiederholt wird. Der reguläre Ausdruck ‹[0-9]{3}› entspricht daher ‹[0-9][0-9][0-9]›, ist aber kürzer und hoffentlich leichter zu lesen. Ein (schon weiter oben erwähntes) Fragezeichen ist ein besonderer Quantor, durch den das vorherige Element genau null oder ein Mal wiederholt werden kann. Er lässt sich auch als ‹{0,1}› schreiben. Jeder Quantor, durch den etwas auch null Mal wiederholt werden kann, sorgt im Endeffekt dafür, dass das Element optional ist. Da hinter jedem Trennzeichen ein Fragezeichen steht, können die Ziffern der Telefonnummer auch ohne Trennzeichen direkt hintereinanderstehen.

Beachten Sie, dass wir zwar von nordamerikanischen Telefonnummern reden, die Regex aber präziser dafür gedacht ist, Telefonnummern nach dem *North American Numbering Plan* (NANP) zu finden. Der NANP ist der Nummerierungsplan für die Länder, die gemeinsam die Ländervorwahl „1“ nutzen. Dazu gehören die USA und seine Territorien, Kanada, die Bermudas und 16 Karibikstaaten. Mexiko und die Staaten in Mittelamerika gehören nicht dazu.

Variationen

Ungültige Telefonnummern eliminieren

Der reguläre Ausdruck passt so weit zu jeder zehnstelligen Nummer. Wenn Sie eine erfolgreiche Suche aber auf Telefonnummern beschränken wollen, die nach dem North American Numbering Plan gültig sind, müssen noch ein paar weitere Regeln beachtet werden:

- *Vorwahlen* beginnen mit einer Ziffer von 2 bis 9, gefolgt von einer Ziffer von 0 bis 8. Die dritte Ziffer kann beliebig sein.
- Die zweite dreistellige Gruppe, die auch als *Central Office* oder *Exchange Code* bekannt ist, beginnt mit einer Ziffer zwischen 2 und 9, gefolgt von zwei weiteren beliebigen Ziffern.
- Für die letzten vier Ziffern, auch bekannt als *Station Code*, gibt es keine Einschränkungen.

Diese Regeln lassen sich leicht mit ein paar Zeichenklassen umsetzen:

```
^\(?([2-9][0-8][0-9])\)?[-.”]?([2-9][0-9]{2})[-.”]?([0-9]{4})$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Neben den eben beschriebenen grundlegenden Regeln gibt es eine Reihe von reservierten, nicht zugewiesenen und eingeschränkten Telefonnummern. Sofern Sie nicht unbedingt so viele Telefonnummern wie möglich aussortieren müssen, sollten Sie es mit dem Entfernen ungenutzter Nummern nicht zu weit treiben. Neue Vorwahlen, die den oben aufgeführten Regeln entsprechen, werden immer wieder mal freigeschaltet, und selbst wenn eine Telefonnummer gültig ist, muss das nicht unbedingt bedeuten, dass sie geschaltet wurde oder aktuell genutzt wird.

Telefonnummern in Dokumenten finden

Durch zwei einfache Änderungen kann der vorherige reguläre Ausdruck auch Telefonnummern in längeren Texten finden:

```
\(?\b([0-9]{3})\)?[-.”]?([0-9]{3})[-.”]?([0-9]{4})\b
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Hier wurden die Zusicherungen ‹^› und ‹$›, mit denen der reguläre Ausdruck mit dem Anfang und Ende des Texts verbunden wurde, entfernt. Stattdessen wurden Wortgrenzen (‹\b›) ergänzt, mit denen sichergestellt wird, dass der gefundene Text für sich steht und nicht Teil einer längeren Zahl oder eines Wortes ist.

Wie ‹^› und ‹$› ist ‹\b› eine Zusicherung, die eine Position anstelle von echtem Text findet. Genauer gesagt, passt ‹\b› zu einer Position zwischen einem Wortzeichen und entweder einem Nicht-Wortzeichen oder dem Anfang oder Ende des Texts. Dabei werden Buchstaben, Ziffern und der Unterstrich als Wortzeichen angesehen (siehe Rezept 2.6).

Beachten Sie, dass die erste Wortgrenze erst nach der optionalen öffnenden Klammer steht. Das ist wichtig, denn es gibt keine Wortgrenze, die man zwischen zwei Nicht-Wortzeichen finden kann – wie zum Beispiel zwischen der öffnenden Klammer und einem vorherigen Leerzeichen. Die erste Wortgrenze ist nur dann wichtig, wenn eine Nummer ohne Klammern gefunden wird. Denn die Wortgrenze passt immer zwischen der öffnenden Klammer und der ersten Ziffer einer Telefonnummer.

Eine führende „1“ zulassen

Sie können eine optionale führende „1“ für den Ländercode zulassen (durch den der Bereich des North American Numbering Plan erreichbar ist), indem Sie die Ergänzungen der folgenden Regex nutzen:

```
^(?:\+?1[-.”]?)?\(?([0-9]{3})\)?[-.”]?([0-9]{3})[-.”]?([0-9]{4})$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Neben den schon gezeigten Telefonnummerformaten passt dieser reguläre Ausdruck auch zu Strings wie `+1 (123) 456-7890` und `1-123-456-7890`. Dabei wird eine nicht-einfangende Gruppe genutzt, die die Syntax ‹(?:···)› besitzt. Wenn auf eine öffnende Klammer ein Fragezeichen folgt, handelt es sich dabei nicht um einen Quantor, sondern damit wird die Art der Gruppe festgelegt. Normale einfangende Gruppen zwingen die Regex-Engine dazu, sich Rückwärtsreferenzen zu merken, daher ist es effizienter, nicht-einfangende Gruppen zu verwenden, wenn der von einer Gruppe gefundene Text später nicht referenziert werden muss. Ein anderer Grund für die Verwendung einer nicht-einfangenden Gruppe ist, die gleichen Ersetzungstexte zu verwenden wie in den vorherigen Beispielen. Hätten wir eine einfangende Gruppe hinzugefügt, müssten wir im weiter oben gezeigten Ersetzungstext `$1` durch `$2` ersetzen (und so weiter).

In dieser Regex wurde ‹(?:\+?1[-.”]?)?› hinzugefügt. Vor der „1“ steht in diesem Muster ein optionales Pluszeichen und dahinter eines von drei Trennzeichen (Bindestrich, Punkt oder Leerzeichen). Die gesamte nicht-einfangende Gruppe ist zudem optional, aber da die „1“ in der Gruppe notwendig ist, dürfen das vorhergehende Pluszeichen und das Trennzeichen nicht erscheinen, wenn es keine „1“ gibt.

Telefonnummern mit sieben Ziffern zulassen

Um auch Telefonnummern zu finden, bei denen die Vorwahl weggelassen wurde, umschließen Sie die erste Zifferngruppe zusammen mit seinen umgebenden Klammern und dem folgenden Trennzeichen mit einer optionalen nicht-einfangenden Gruppe:

```
^(?:\(?([0-9]{3})\)?[-.”]?)?([0-9]{3})[-.”]?([0-9]{4})$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Da die Vorwahl nun nicht mehr notwendigerweise Teil der Übereinstimmung ist, würde ein einfaches Ersetzen durch `«($1)”$2-$3»` nun eventuell zu so etwas wie `() 123-4567` führen. Um das zu umgehen, sollten Sie die Regex zusammen mit Code verwenden, der prüft, ob in der ersten Gruppe überhaupt Text enthalten ist. Wenn nicht, muss der Ersetzungstext dementsprechend angepasst werden.

Siehe auch

In Rezept 4.3 wird gezeigt, wie Sie internationale Telefonnummern überprüfen.

Der North American Numbering Plan (NANP) ist der Nummernplan für Telefonanschlüsse in den USA und seinen Territorien, in Kanada, auf den Bermudas und in 16 karibischen Staaten. Unter *http://www.nanpa.com* erhalten Sie mehr Informationen dazu.

4.3 Internationale Telefonnummern überprüfen

Problem

Sie wollen internationale Telefonnummern überprüfen. Die Nummern sollen mit einem Pluszeichen beginnen, auf das die Ländervorwahl und dann die Nummer innerhalb des Landes folgt.

Lösung

Regulärer Ausdruck

```
^\+(?:[0-9]”?){6,14}[0-9]$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

JavaScript

```
function validate (phone) {
    var regex = /^\+(?:[0-9] ?){6,14}[0-9]$/;

    if (regex.test(phone)) {
        // Gültige internationale Telefonnummer
    } else {
        // Ungültige internationale Telefonnummer
    }
}
```

Andere Programmiersprachen

In Rezept 3.5 finden Sie Informationen, wie dieser reguläre Ausdruck mit anderen Programmiersprachen implementiert werden kann.

Diskussion

Die Regeln und Richtlinien für die Ausgabe internationaler Telefonnummern unterscheiden sich von Land zu Land, daher ist es schwierig, für solche Nummern eine sinnvolle Validierung durchzuführen, wenn Sie sich nicht gerade an ein striktes Format halten. Zum Glück gibt es eine einfache Standardnotation, die durch die ITU-T E.123 definiert ist. Dabei gehört zu internationalen Telefonnummern ein führendes Pluszeichen (auch als *International Prefix Symbol* bezeichnet), und als Trennzeichen sind nur Leerzeichen zugelassen, um die Zifferngruppen zu unterteilen. Auch wenn das Tildezeichen (~) innerhalb einer Telefonnummer auftauchen kann, um auf einen zusätzlichen Wählton hinzuweisen, haben wir dieses nicht in den regulären Ausdruck integriert, da es sich eher um ein prozedurales Element handelt (es wird also nicht mitgewählt) und auch nur sehr

selten zur Anwendung kommt. Aufgrund des internationalen Telefonnummernplans (ITU-T E.164) können Telefonnummern mehr als 15 Ziffern enthalten. Die kürzeste internationale Telefonnummer, die auch genutzt wird, besteht aus sieben Ziffern.

Mit all dem im Hinterkopf wollen wir uns den regulären Ausdruck anschauen, nachdem wir ihn in seine Bestandteile zerlegt haben. Da diese Version im Freiform-Modus geschrieben ist, wurden die literalen Leerzeichen durch ‹\x20› ersetzt:

```
^         # Sicherstellen, dass die Übereinstimmung am Anfang des Texts beginnt.
\+        # Ein literales Pluszeichen finden.
(?:       # Gruppieren, aber nicht einfangen ...
  [0-9]   #   Eine Ziffer finden.
  \x20    #   Ein Leerzeichen finden ...
    ?     #     null oder ein Mal.
)         # Ende der nicht-einfangenden Gruppe.
  {6,14}  #   Wiederholen der vorherigen Gruppe 6 bis 14 Mal.
[0-9]     # Eine Ziffer finden.
$         # Sicherstellen, dass die Übereinstimmung am Ende des Texts endet.
```

Regex-Optionen: Freiform
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Die Anker ‹^› und ‹$› am Anfang und am Ende des regulären Ausdrucks stellen sicher, dass der gesamte Ausgangstext passen muss. Die nicht-einfangende Gruppe – umschlossen von ‹(?:···)› – passt zu einer einzelnen Ziffer, gefolgt von einem optionalen Leerzeichen. Durch das Wiederholen dieser Gruppe mit einem Intervall-Quantor ‹{6,14}› werden die minimale und die maximale Anzahl an Ziffern festgelegt, während gleichzeitig an beliebiger Stelle in der Nummer auch Leerzeichen stehen dürfen. Die zweite Instanz der Zeichenklasse ‹[0-9]› vervollständigt die Regel für die Anzahl der Ziffern (von 6 bis 14 auf 7 bis 15) und stellt sicher, dass die Telefonnummer nicht mit einem Leerzeichen endet.

Variationen

Validieren von internationalen Telefonnummern im EPP-Format

```
^\+[0-9]{1,3}\.[0-9]{4,14}(?:x.+)?$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Dieser reguläre Ausdruck orientiert sich an der Notation für internationale Telefonnummern, die durch das Extensible Provisioning Protocol (EPP) definiert ist. EPP ist ein recht neues Protokoll (es wurde 2004 abgeschlossen) und dient dazu, die Kommunikation zwischen Domain Name Registries und Registraren zu unterstützen. Es wird von einer wachsenden Zahl von Domain Name Registries genutzt, so für *.com*, *.info*, *.net*, *.org* und *.us*. Entscheidend dabei ist, dass internationale Telefonnummern im EPP-Stil zunehmend verwendet und auch erkannt werden. Damit stellt es ein gutes alternatives Format für das Speichern (und Validieren) internationaler Telefonnummern bereit.

Telefonnummern im EPP-Stil verwenden das Format *+CCC.NNNNNNNNNxEEEE*, wobei *C* der ein- bis dreistellige Ländercode ist, *N* aus bis zu 14 Ziffern bestehen kann und *E* eine (optionale) Durchwahl ist. Das führende Pluszeichen und der Punkt, der auf den Ländercode folgt, sind verpflichtend. Das literale „x" ist nur dann notwendig, wenn eine Durchwahl angegeben wird.

Siehe auch

In Rezept 4.2 werden noch mehr Möglichkeiten für das Validieren nordamerikanischer Telefonnummern vorgestellt.

Die ITU-T-Empfehlung E.123 (»Notation for national and international telephone numbers, e-mail addresses and Web addresses«) kann unter *http://www.itu.int/rec/T-REC-E.123* heruntergeladen werden.

Die ITU-T-Empfehlung E.164 (»The international public telecommunication numbering plan«) kann unter *http://www.itu.int/rec/T-REC-E.164* heruntergeladen werden.

Nationale Nummernpläne lassen sich unter *http://www.itu.int/ITU-T/inr/nnp* herunterladen.

Der RFC 4933 definiert die Syntax und Semantik der EPP Contact Identifiers, zu denen auch die internationalen Telefonnummern gehören. Sie können den RFC 4933 unter *http://tools.ietf.org/html/rfc4933* herunterladen.

4.4 Klassische Datumsformate validieren

Problem

Sie wollen Datumswerte in den klassischen Formaten mm/dd/yy, mm/dd/yyyy, dd.mm.yy und dd.mm.yyyy validieren. Dafür wollen Sie eine einfache Regex verwenden, die nur prüft, ob ein Wert wie ein Datum aussieht, ohne aber ungültige Datumswerte wie den 31. Februar zu erkennen.

Lösung

Finden eines dieser Datumsformate, wobei führende Nullen weggelassen werden dürfen:

```
^[0-3]?[0-9][/.][0-3]?[0-9][/.](?:[0-9]{2})?[0-9]{2}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Finden eines dieser Datumsformate, wobei führende Nullen vorhanden sein müssen:

```
^[0-3][0-9][/.][0-3][0-9][/.](?:[0-9][0-9])?[0-9][0-9]$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Finden von m/d/yy und mm/dd/yyyy, wobei jede Kombination aus einer oder zwei Ziffern für Tag und Monat zulässig sind und zwei oder vier Ziffern für das Jahr:

```
^(1[0-2]|0?[1-9])/(3[01]|[12][0-9]|0?[1-9])/(?:[0-9]{2})?[0-9]{2}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Finden von mm/dd/yyyy, führende Nullen müssen vorhanden sein:

```
^(1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])/[0-9]{4}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Finden von d.m.yy und dd.mm.yyyy, wobei jede Kombination aus einer oder zwei Ziffern für Tag und Monat zulässig sind und zwei oder vier Ziffern für das Jahr:

```
^(3[01]|[12][0-9]|0?[1-9])\.(1[0-2]|0?[1-9])\.(?:[0-9]{2})?[0-9]{2}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Finden von dd.mm.yyyy, führende Nullen müssen vorhanden sein:

```
^(3[01]|[12][0-9]|0[1-9])\.(1[0-2]|0[1-9])\.[0-9]{4}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Finden eines dieser Datumsformate mit besserer Genauigkeit, wobei führende Nullen weggelassen werden dürfen:

```
^(?:(1[0-2]|0?[1-9])/(3[01]|[12][0-9]|0?[1-9])/|: "
(3[01]|[12][0-9]|0?[1-9])\.(1[0-2]|0?[1-9])\.)(?:[0-9]{2})?[0-9]{2}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Finden eines dieser Datumsformate mit besserer Genauigkeit, führende Nullen müssen vorhanden sein:

```
^(?:(1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])/|: "
(3[01]|[12][0-9]|0[1-9])\.(1[0-2]|0[1-9])\.)[0-9]{4}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Im Freiform-Modus sind die letzten beiden Regexes etwas besser lesbar:

```
^(?:
  # m/d/ oder mm/dd/
  (1[0-2]|0?[1-9])/(3[01]|[12][0-9]|0?[1-9])/
|
  # d.m. oder dd.mm.
```

```
  (3[01]|[12][0-9]|0?[1-9])\.(1[0-2]|0?[1-9])\.
)
# yy oder yyyy
(?:[0-9]{2})?[0-9]{2}$
```

Regex-Optionen: Freiform
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:
  # mm/dd/
  (1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])
|
  # dd.mm.
  (3[01]|[12][0-9]|0[1-9])\.(1[0-2]|0[1-9])\.
)
# yyyy
[0-9]{4}$
```

Regex-Optionen: Freiform
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Diskussion

Vielleicht gehen Sie davon aus, dass etwas konzeptionell so Einfaches wie ein Datumswert für einen regulären Ausdruck ein leichtes Spiel wäre. Aber dem ist nicht so, und zwar aus zwei Gründen. Da Datumswerte tagtäglich verwendet werden, gehen Menschen mit ihnen sehr salopp um. `4/1` ist für einen Amerikaner vielleicht der 1. April. Für jemand anderen ist es dagegen eventuell der erste Arbeitstag im Jahr, wenn Neujahr auf einen Freitag fällt. Die gezeigten Lösungen passen zu einigen der am häufigsten genutzten Datumsformaten.

Das andere Problem ist, dass reguläre Ausdrücke nicht direkt mit Zahlen arbeiten. Sie können einen regulären Ausdruck nicht anweisen, zum Beispiel „eine Zahl zwischen 1 und 31 zu finden“. Reguläre Ausdrücke gehen Zeichen für Zeichen vor. Mit `‹3[01]|[12][0-9]|0?[1-9]›` finden wir eine 3, gefolgt von einer 0 oder 1, oder eine 1 oder 2, gefolgt von einer Ziffer, oder eine optionale 0, gefolgt von einer Ziffer zwischen 1 und 9. Bei Zeichenklassen können wir für einzelne Ziffern Bereiche nutzen, wie zum Beispiel `‹[1-9]›`. Das liegt daran, dass die Zeichen für die Ziffern 0 bis 9 in den Zeichentabellen von ASCII und Unicode direkt aufeinanderfolgen. In Kapitel 6 finden Sie mehr Details über das Finden aller möglichen Arten von Zahlen mit regulären Ausdrücken.

Aus diesen Gründen müssen Sie sich entscheiden, wie einfach oder wie genau Ihr regulärer Ausdruck sein soll. Wenn Sie schon wissen, dass der Ausgangstext keine ungültigen Datumswerte enthält, können Sie eine einfache Regex wie `‹\d{2}/\d{2}/\d{4}›` oder `‹\d{2}\.\d{2}\.\d{4}›` nutzen. Damit wird zwar auch so etwas wie `99/99/9999` gefunden, aber das ist unwichtig, weil Sie ja wissen, dass solche Werte im Ausgangstext nicht vorkommen. Sie können diese einfache Regex schnell eingeben, und sie wird auch schnell ausgewertet.

Die ersten beiden Lösungen für dieses Rezept sind ebenfalls schnell und einfach, und sie passen auch zu ungültigen Datumswerten wie `0/0/00` und `31/31/2008`. Dabei werden für die Trennzeichen und für die Ziffern Zeichenklassen genutzt (siehe Rezept 2.3) sowie das Fragezeichen (siehe Rezept 2.12), um bestimmte Ziffern optional zu machen. Mit `‹(?:[0-9]{2})?[0-9]{2}›` kann das Jahr aus zwei oder vier Ziffern bestehen. `‹[0-9]{2}›` passt genau zu zwei Ziffern, `‹(?:[0-9]{2})?›` passt zu null oder zwei Ziffern. Die nicht-einfangende Gruppe (siehe Rezept 2.9) ist hier erforderlich, da das Fragezeichen auf die Kombination aus Zeichenklasse und Quantor `‹{2}›` angewandt werden muss. `‹[0-9]{2}?›` passt wie `‹[0-9]{2}›` genau zu zwei Ziffern. Ohne die Gruppe würde das Fragezeichen dafür sorgen, dass der Quantor genügsam wird, was hier keinen Effekt hat, da `‹{2}›` nicht mehr oder weniger als zwei Mal zu einer Wiederholung führen kann.

Die Lösungen 3 bis 6 grenzen den Monat auf Zahlen zwischen 1 und 12 und den Tag auf Werte zwischen 1 und 31 ein. Mithilfe der Alternation (siehe Rezept 2.8) innerhalb einer Gruppe finden wir einen bestimmten Bereich von zweistelligen Zahlen. Hier verwenden wir einfangende Gruppen, weil Sie Tag und Monat vermutlich sowieso auslesen wollen.

Die letzten beiden Lösungen sind ein bisschen komplexer, daher zeigen wir sie auch im Freiform-Modus. Der einzige Unterschied zwischen den beiden Formen ist die Lesbarkeit. In JavaScript wird allerdings kein Freiform-Modus unterstützt. Diese letzten Lösungen ermöglichen alle Datumsformate – so wie die ersten beiden Lösungen. Aber hier wird durch eine zusätzliche Alternation dafür gesorgt, dass Datumswerte auf 12/31 und 31.12. begrenzt werden und somit keine ungültigen Monate (31/31) möglich sind.

Variationen

Wenn Sie in einem umfangreicheren Text nach einem Datumswert suchen wollen, statt eine Benutzereingabe als Ganzes zu prüfen, können Sie die Anker `‹^›` und `‹$›` nicht verwenden. Sie einfach zu entfernen, wäre aber auch nicht die richtige Lösung. Damit würde zum Beispiel in `9912/12/200199` der Wert `12/12/2001` gefunden werden. Statt die Regex-Übereinstimmung mit dem Anfang und dem Ende des Texts zu verankern, müssen Sie festlegen, dass das Datum kein Teil einer längeren Ziffernfolge sein darf.

Das lässt sich ganz einfach mit Wortgrenzen erreichen. In regulären Ausdrücken werden Ziffern als Wortzeichen behandelt. Ersetzen Sie sowohl `‹^›` als auch `‹$›` durch `‹\b›`. Ein Beispiel:

```
\b(1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])/[0-9]{4}\b
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Siehe auch

Rezepte 4.5, 4.6 und 4.7.

4.5 Klassische Datumsformate exakt validieren

Problem

Sie wollen Datumswerte in den klassischen Formaten mm/dd/yy, mm/dd/yyyy, dd.mm.yy und dd.mm.yyyy validieren. Dabei wollen Sie aber ungültige Datumswerte wie den 31. Februar ausschließen.

Lösung

C#

Monat vor Tag:

```
DateTime foundDate;
Match matchResult = Regex.Match(SubjectString,
    "^(?<month>[0-3]?[0-9])/(?<day>[0-3]?[0-9])/" +
    "(?<year>(?:[0-9]{2})?[0-9]{2})$");
if (matchResult.Success) {
    int year = int.Parse(matchResult.Groups["year"].Value);
    if (year < 50) year += 2000;
    else if (year < 100) year += 1900;
    try {
        foundDate = new DateTime(year,
            int.Parse(matchResult.Groups["month"].Value),
            int.Parse(matchResult.Groups["day"].Value));
    } catch {
        // Ungültiges Datum
    }
}
```

Tag vor Monat:

```
DateTime foundDate;
Match matchResult = Regex.Match(SubjectString,
    "^(?<day>[0-3]?[0-9])\.(?<month>[0-3]?[0-9])\." +
    "(?<year>(?:[0-9]{2})?[0-9]{2})$");
if (matchResult.Success) {
    int year = int.Parse(matchResult.Groups["year"].Value);
    if (year < 50) year += 2000;
    else if (year < 100) year += 1900;
    try {
        foundDate = new DateTime(year,
            int.Parse(matchResult.Groups["month"].Value),
            int.Parse(matchResult.Groups["day"].Value));
    } catch {
        // Ungültiges Datum
    }
}
```

Perl

Monat vor Tag:

```
@daysinmonth = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
$validdate = 0;
if ($subject =~ m!^([0-3]?[0-9])/([0-3]?[0-9])/((?:[0-9]{2})?[0-9]{2})$!) {
    $month = $1;
    $day = $2;
    $year = $3;
    $year += 2000 if $year < 50;
    $year += 1900 if $year < 100;
    if ($month == 2 && $year % 4 == 0 && ($year % 100 != 0 ||
                                          $year % 400 == 0)) {
        $validdate = 1 if $day >= 1 && $day <= 29;
    } elsif ($month >= 1 && $month <= 12) {
        $validdate = 1 if $day >= 1 && $day <= $daysinmonth[$month-1];
    }
}
```

Tag vor Monat:

```
@daysinmonth = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
$validdate = 0;
if ($subject =~ m!^([0-3]?[0-9])\.([0-3]?[0-9])\.((?:[0-9]{2})?[0-9]{2})$!) {
    $day = $1;
    $month = $2;
    $year = $3;
    $year += 2000 if $year < 50;
    $year += 1900 if $year < 100;
    if ($month == 2 && $year % 4 == 0 && ($year % 100 != 0 ||
                                          $year % 400 == 0)) {
        $validdate = 1 if $day >= 1 && $day <= 29;
    } elsif ($month >= 1 && $month <= 12) {
        $validdate = 1 if $day >= 1 && $day <= $daysinmonth[$month-1];
    }
}
```

Lösung nur mit einem regulären Ausdruck:

Monat vor Tag:

```
^(?:
  # Februar (jedes Jahr 29 Tage)
  (?<month>0?2)/(?<day>[12][0-9]|0?[1-9])
|
  # Monate mit 30 Tagen
  (?<month>0?[469]|11)/(?<day>30|[12][0-9]|0?[1-9])
|
  # Monate mit 31 Tagen
  (?<month>0?[13578]|1[02])/(?<day>3[01]|[12][0-9]|0?[1-9])
)
# Jahr
/(?<year>(?:[0-9]{2})?[0-9]{2})$
```

Regex-Optionen: Freiform
Regex-Varianten: .NET

```
^(?:
  # Februar (jedes Jahr 29 Tage)
  (0?2)/([12][0-9]|0?[1-9])
|
  # Monate mit 30 Tagen
  (0?[469]|11)/(30|[12][0-9]|0?[1-9])
|
  # Monate mit 31 Tagen
  (0?[13578]|1[02])/(3[01]|[12][0-9]|0?[1-9])
)
# Jahr
/((?:[0-9]{2})?[0-9]{2})$
```

Regex-Optionen: Freiform

Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:(0?2)/([12][0-9]|0?[1-9])|(0?[469]|11)/(30|[12][0-9]|0?[1-9])|: "
(0?[13578]|1[02])/(3[01]|[12][0-9]|0?[1-9]))/((?:[0-9]{2})?[0-9]{2})$
```

Regex-Optionen: Keine

Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Tag vor Monat:

```
^(?:
  # Februar (jedes Jahr 29 Tage)
  (?<day>[12][0-9]|0?[1-9])\.(?<month>0?2)
|
  # Monate mit 30 Tagen
  (?<day>30|[12][0-9]|0?[1-9])\.(?<month>0?[469]|11)
|
  # Monate mit 31 Tagen
  (?<day>3[01]|[12][0-9]|0?[1-9])\.(?<month>0?[13578]|1[02])
)
# Jahr
\.(?<year>(?:[0-9]{2})?[0-9]{2})$
```

Regex-Optionen: Freiform

Regex-Varianten: .NET

```
^(?:
  # Februar (jedes Jahr 29 Tage)
  ([12][0-9]|0?[1-9])\.(0?2)
|
  # Monate mit 30 Tagen
  (30|[12][0-9]|0?[1-9])\.([469]|11)
|
  # Monate mit 31 Tagen
  (3[01]|[12][0-9]|0?[1-9])\.(0?[13578]|1[02])
)
# Jahr
\.((?:[0-9]{2})?[0-9]{2})$
```

Regex-Optionen: Freiform

Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:([12][0-9]|0?[1-9])\.(0?2)|(30|[12][0-9]|0?[1-9])\.([469]|11)|: "
(3[01]|[12][0-9]|0?[1-9])\.(0?[13578]|1[02]))\.((?:[0-9]{2})?[0-9]{2})$
```

Regex-Optionen: Keine

Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Diskussion

Es gibt im Prinzip zwei Wege, Datumswerte mit einem regulären Ausdruck exakt zu validieren. Der eine ist, eine einfache Regex zu verwenden, die eigentlich nur die Zahlengruppen einfängt, die wie eine Kombination aus Monat/Tag/Jahr beziehungsweise Tag.Monat.Jahr aussehen, und dann mit prozeduralem Code zu prüfen, ob das Datum korrekt ist. Ich habe die erste Regex aus dem vorhergehenden Rezept verwendet, die eine beliebige Zahl zwischen 0 und 39 für Tag und Monat zuließ. Damit lässt sich das Format leicht zwischen mm/dd/yy und dd.mm.yy wechseln, indem man sich aussucht, welche Gruppe den Tag und welche den Monat einfängt (abgesehen vom unterschiedlichen Trennzeichen).

Der Hauptvorteil dieser Methode ist, dass Sie einfach zusätzliche Einschränkungen ergänzen können, wie zum Beispiel ein Einschränken auf bestimmte Zeiträume. Viele Programmiersprachen stellen eine ganze Reihe von Möglichkeiten bereit, mit Datumswerten zu arbeiten. Die C#-Lösung verwendet die .NET-Struktur `DateTime`, um zu prüfen, ob das Datum gültig ist, und es in einem sinnvollen Format zurückzugeben.

Der andere Weg ist, alles mit einem regulären Ausdruck abzudecken. Das lässt sich dann erreichen, wenn wir uns die Freiheit nehmen, jedes Jahr als Schaltjahr zu betrachten. Wir können die gleiche Technik für die Alternativen verwenden, die wir schon im letzten Rezept genutzt haben.

Mit einem einzelnen regulären Ausdruck haben wir aber nun das Problem, dass Tag und Monat nicht mehr übersichtlich in einer einfangenden Gruppe erfasst werden. Jetzt gibt es drei einfangende Gruppen für den Monat und drei für den Tag. Passt die Regex auf ein Datum, enthalten nur drei der sieben Gruppen in der Regex etwas. Handelt es sich um ein Datum im Februar, fangen die Gruppen 1 und 2 den Monat und den Tag (beziehungsweise umgekehrt). Hat der Monat 30 Tage, enthalten die Gruppen 3 und 4 diese Werte. Bei Monaten mit 31 Tagen muss man sich die Gruppen 5 und 6 anschauen. In Gruppe 7 ist immer das Jahr zu finden.

In dieser Situation hilft uns nur die .NET-Variante. Hier können verschiedene benannte einfangende Gruppen (siehe Rezept 2.11) den gleichen Namen tragen und den gleichen Speicherplatz für ihren Wert verwenden. Nutzen Sie die .NET-Lösung mit benannten Captures, können Sie den Text, der durch die Gruppen „month“ und „day“ gefunden wurden, einfach auslesen, ohne sich darum Gedanken machen zu müssen, wie viele Tage der Monat hat. Alle anderen in diesem Buch behandelten Varianten lassen es nicht zu, dass zwei Gruppen den gleichen Namen tragen, oder sie geben nur den Text zurück, der von der letzten einfangenden Gruppe mit einem bestimmten Namen gefunden wurde. Bei diesen Varianten kann man nur mit nummerierten Captures arbeiten.

Die Lösung, die mit lediglich einer Regex arbeitet, ist nur dann interessant, wenn Sie auch nur eine Regex verwenden können – beispielsweise in dem Fall, dass eine Anwendung nur ein Eingabefeld für eine Regex anbietet. Beim Programmieren wird alles viel einfacher, wenn Sie ein bisschen Code darum herum bauen. Das ist vor allem dann hilfreich, wenn Sie die Datumswerte später noch verarbeiten wollen. Hier finden Sie eine reine Regex-Lösung, die ein Datum zwischen dem 2. Mai 2007 und dem 29. August 2008 im Format d.m.yy oder dd.mm.yyyy findet:

```
# 2. Mai 2007 bis 29. August 2008
^(?:
  # 2. Mai 2007 bis 31. Dezember 2007
  (?:
    # 2. Mai bis 31. Mai
    (?<day>3[01]|[12][0-9]|0?[2-9])\.(?<month>0?5)\.(?<year>2007)
  |
    # 1. Juni bis 31. Dezember
    (?:
      # Monate mit 30 Tagen
      (?<day>30|[12][0-9]|0?[1-9])\.(?<month>0?[69]|11)
    |
      # Monate mit 31 Tagen
      (?<day>3[01]|[12][0-9]|0?[1-9])\.(?<month>0?[78]|1[02])
    )
    \.(?<year>2007)
  )
|
  # 1. Januar 2008 bis 29. August 2008
  (?:
    # 1. August bis 29. August
    (?<day>[12][0-9]|0?[1-9])\.(?<month>0?8)\.(?<year>2008)
  |
    # 1. Januar bis 31. Juli
    (?:
      # Februar
      (?<day>[12][0-9]|0?[1-9])\.(?<month>0?2)
    |
      # Monate mit 30 Tagen
      (?<day>30|[12][0-9]|0?[1-9])\.(?<month>0?[46])
    |
      # Monate mit 31 Tagen
      (?<day>3[01]|[12][0-9]|0?[1-9])\.(?<month>0?[1357])
    )
    \.(?<year>2008)
  )
)$
```

Regex-Optionen: Freiform
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Siehe auch

Rezepte 4.4, 4.6 und 4.7.

4.6 Klassische Zeitformate validieren

Problem

Sie wollen Zeitwerte in klassischen Formaten validieren. Dazu gehören hh:mm und hh:mm:ss – sowohl im 12-Stunden- als auch im 24-Stunden-Format.

Lösung

Stunden und Minuten, 12 Stunden:

```
^(1[0-2]|0?[1-9]):([0-5]?[0-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Stunden und Minuten, 24 Stunden:

```
^(2[0-3]|[01]?[0-9]):([0-5]?[0-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Stunden, Minuten und Sekunden, 12 Stunden:

```
^(1[0-2]|0?[1-9]):([0-5]?[0-9]):([0-5]?[0-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Stunden, Minuten und Sekunden, 24 Stunden:

```
^(2[0-3]|[01]?[0-9]):([0-5]?[0-9]):([0-5]?[0-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Die Fragezeichen in allen angegebenen regulären Ausdrücken sorgen dafür, dass führende Nullen optional werden. Sollen sie immer angegeben werden, entfernen Sie die Fragezeichen.

Diskussion

Das Validieren von Uhrzeiten ist deutlich einfacher als das Validieren von Datumswerten. Jede Stunde hat 60 Minuten und jede Minute 60 Sekunden. Somit brauchen wir keine komplizierten Alternationen in der Regex. Für Minuten und Sekunden sind sogar überhaupt keine Alternationen notwendig. `‹[0-5]?[0-9]›` passt zu einer Ziffer zwischen 0 und 5, gefolgt von einer Ziffer zwischen 0 und 9. Damit werden alle Zahlen zwischen 0 und 59 gefunden. Durch das Fragezeichen nach der ersten Zeichenklasse wird diese optional. So wird auch eine einzelne Ziffer zwischen 0 und 9 als gültige Minute oder Sekunde

erkannt. Entfernen Sie das Fragezeichen, wenn die ersten zehn Minuten und Sekunden als 00 bis 09 geschrieben werden sollen. In Rezept 2.3 und Rezept 2.12 finden Sie Details zu Zeichenklassen und Quantoren.

Bei den Stunden brauchen wir dann aber eine Alternation (siehe Rezept 2.8). Für die zweite Ziffer sind abhängig von der ersten Ziffer unterschiedliche Bereiche notwendig. Bei einer 12-Stunden-Uhr sind für die zweite Ziffer alle 10 Ziffern erlaubt, wenn die erste Ziffer eine 0 ist. Ist die erste Ziffer dagegen eine 1, muss die zweite Ziffer entweder 0, 1 oder 2 sein. Bei einem regulären Ausdruck schreiben wir das als ‹1[0-2]|0?[1-9]›. Bei einer 24-Stunden-Uhr sind alle zehn Ziffern für die zweite Ziffer erlaubt, wenn die erste Ziffer 0 oder 1 ist. Ist diese aber 2, muss die zweite Ziffer zwischen 0 und 3 liegen. In Regex-Syntax lässt sich das als ‹2[0-3]|[01]?[0-9]› darstellen. Auch hier dient das Fragezeichen wieder dazu, dass die ersten zehn Stunden als einzelne Ziffer geschrieben werden dürfen. Wenn Sie es entfernen, müssen immer zwei Ziffern angegeben werden.

Wir haben die Teile der Regex, mit der die Stunden, Minuten und Sekunden gefunden werden, mit Klammern versehen. Dadurch ist es einfach, die Ziffern ohne die Trennzeichen auszulesen. In Rezept 2.9 ist erklärt, wie Klammern für einfangende Gruppen genutzt werden. Und in Rezept 3.9 wird beschrieben, wie Sie den von solchen einfangenden Gruppen gefundenen Text mithilfe von prozeduralem Code auslesen können.

Die Klammern um den Stundenteil sorgen dafür, dass die zwei Alternativen für die Stunden zusammengehalten werden. Entfernen Sie die Klammern, wird die Regex nicht mehr richtig funktionieren. Entfernen Sie die Klammern um die Minuten und Sekunden, hat das keine Auswirkungen, außer dass Sie dann die Ziffern nicht mehr einzeln auslesen können.

Variationen

Wenn Sie in längeren Texten nach Uhrzeiten suchen wollen, statt zu prüfen, ob eine Eingabe nur aus einer Zeitangabe besteht, können Sie die Anker ‹^› und ‹$› nicht nutzen. Es reicht aber auch nicht aus, die Anker einfach zu entfernen. Damit würden Regexes für Stunden und Minuten den Wert 12:12 innerhalb von 9912:1299 finden. Statt den Anfang und das Ende des Texts mit Anfang und Ende der Regex zu verknüpfen, müssen Sie festlegen, dass die Uhrzeit kein Teil einer längeren Folge von Ziffern sein kann.

Das lässt sich leicht mit einem Paar Wortgrenzen erreichen. In regulären Ausdrücken werden Ziffern als Wortzeichen behandelt. Ersetzen Sie also sowohl ‹^› als auch ‹$› durch ‹\b›, beispielsweise:

```
\b(2[0-3]|[01]?[0-9]):([0-5]?[0-9])\b
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Wortgrenzen verhindern nicht alles – nur Buchstaben, Ziffern und den Unterstrich. Die hier gezeigte Regex passt zu Stunden und Minuten auf einer 24-Stunden-Uhr, findet aber auch 16:08 im Text `Es ist jetzt genau 16:08:42`. Das Leerzeichen ist kein Wortbuch-

stabe, die `1` aber schon, daher passt die Wortgrenze dazwischen. Die `8` ist ein Wortzeichen, der Doppelpunkt aber nicht, also passt `‹\b›` auch hier zwischen.

Wenn Sie sowohl Doppelpunkte als auch Wortzeichen verhindern wollen, müssen Sie ein Lookaround einsetzen (siehe Rezept 2.16). Die folgende Regex findet den Uhrzeit-Teil von `Es ist jetzt genau 16:08:42` nicht. Allerdings funktioniert sie nur mit Varianten, die Lookbehinds zulassen:

```
(?<![:\w])(2[0-3]|[01]?[0-9]):([0-5]?[0-9])(?![:\w])
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Siehe auch

Rezepte 4.4, 4.5 und 4.7.

4.7 Datums- und Uhrzeitwerte im Format ISO 8601 validieren

Problem

Sie wollen Datums- und/oder Uhrzeitwerte im offiziellen Format ISO 8601 finden. Dieses Format ist die Grundlage vieler standardisierter Datums- und Zeitformate. So basieren zum Beispiel in XML Schema die eingebauten Typen `date`, `time` und `dateTime` auf ISO 8601.

Lösung

Die folgende Regex findet einen Kalendermonat, zum Beispiel `2008-08`. Der Bindestrich ist dabei verpflichtend:

```
^([0-9]{4})-(1[0-2]|0[1-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-(?<month>1[0-2]|0[1-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
^(?P<year>[0-9]{4})-(?P<month>1[0-2]|0[1-9])$
```

Regex-Optionen: Keine
Regex-Varianten: PCRE, Python

Datum, zum Beispiel `2008-08-30`. Die Bindestriche sind optional. Diese Regex erlaubt allerdings YYYY-MMDD und YYYYMM-DD, was nicht ISO 8601 entspricht:

```
^([0-9]{4})-?(1[0-2]|0[1-9])-?(3[0-1]|0[1-9]|[1-2][0-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?(?<month>1[0-2]|0[1-9])-?:"
(?<day>3[0-1]|0[1-9]|[1-2][0-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Datum, zum Beispiel `2008-08-30`. Die Bindestriche sind optional. Diese Regex nutzt eine Bedingung, um YYYY-MMDD und YYYYMM-DD auszuschließen. Es gibt eine zusätzliche einfangende Gruppe für den ersten Bindestrich:

```
^([0-9]{4})(-)?(1[0-2]|0[1-9])(?(2)-)(3[0-1]|0[1-9]|[1-2][0-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, PCRE, Perl, Python

Datum, zum Beispiel `2008-08-30`. Die Bindestriche sind optional. Diese Regex nutzt eine Alternation, um YYYY-MMDD und YYYYMM-DD auszuschließen. Es gibt zwei einfangende Gruppen für den Monat:

```
^([0-9]{4})(?:(1[0-2]|0[1-9])|-?(1[0-2]|0[1-9])-?):"
(3[0-1]|0[1-9]|[1-2][0-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Kalenderwoche, zum Beispiel `2008-W35`. Der Bindestrich ist optional:

```
^([0-9]{4})-?W(5[0-3]|[1-4][0-9]|0[1-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?W(?<week>5[0-3]|[1-4][0-9]|0[1-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Tag einer Woche, zum Beispiel `2008-W35-6`. Die Bindestriche sind optional:

```
^([0-9]{4})-?W(5[0-3]|[1-4][0-9]|0[1-9])-?([1-7])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?W(?<week>5[0-3]|[1-4][0-9]|0[1-9])-?(?<day>[1-7])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Tag eines Jahres, zum Beispiel `2008-243`. Der Bindestrich ist optional:

```
^([0-9]{4})-?(36[0-6]|3[0-5][0-9]|[12][0-9]{2}|0[1-9][0-9]|00[1-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?: "
(?<day>36[0-6]|3[0-5][0-9]|[12][0-9]{2}|0[1-9][0-9]|00[1-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Stunden und Minuten, zum Beispiel `17:21`. Der Doppelpunkt ist optional:

```
^(2[0-3]|[01]?[0-9]):?([0-5]?[0-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[01]?[0-9]):?(?<minute>[0-5]?[0-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Stunden, Minuten und Sekunden, zum Beispiel `17:21:59`. Die Doppelpunkte sind optional:

```
^(2[0-3]|[01]?[0-9]):?([0-5]?[0-9]):?([0-5]?[0-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[01]?[0-9]):?(?<minute>[0-5]?[0-9]):?: "
(?<second>[0-5]?[0-9])$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Zeitzonenangabe, zum Beispiel `Z`, `+07` oder `+07:00`. Der Doppelpunkt und die Minuten sind optional:

```
^(Z|[+-](?:2[0-3]|[01]?[0-9])(?::?(?:[0-5]?[0-9]))?)$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Stunden, Minuten und Sekunden mit Zeitzonenangabe, zum Beispiel `17:21:59+07:00`. Alle Doppelpunkte sind optional. Die Minuten in der Zeitzonenangabe sind ebenfalls optional:

```
^(2[0-3]|[01]?[0-9]):?([0-5]?[0-9]):?([0-5]?[0-9]): "
(Z|[+-](?:2[0-3]|[01]?[0-9])(?::?(?:[0-5]?[0-9]))?)$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[01]?[0-9]):?(?<minute>[0-5]?[0-9]):?(?<sec>[0-5]?[0-9]): "
(?<timezone>Z|[+-](?:2[0-3]|[01]?[0-9])(?::?(?:[0-5]?[0-9]))?)$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Datum mit optionaler Zeitzone, zum Beispiel `2008-08-30` oder `2008-08-30+07:00`. Bindestriche sind erforderlich. Das ist der XML Schema-Typ `date`:

```
^(-?(?:[1-9][0-9]*)?[0-9]{4})-(1[0-2]|0[1-9])-(3[0-1]|0[1-9]|[1-2][0-9])": "
(Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Regex-Optionen: Keine

Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>-?(?:[1-9][0-9]*)?[0-9]{4})-(?<month>1[0-2]|0[1-9])-": "
(?<day>3[0-1]|0[1-9]|[1-2][0-9])": "
(?<timezone>Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Regex-Optionen: Keine

Regex-Varianten: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Uhrzeit mit optionalen Sekundenbruchteilen und Zeitzone, zum Beispiel `01:45:36` oder `01:45:36.123+07:00`. Das ist der XML Schema-Typ `time`:

```
^(2[0-3]|[0-1][0-9]):([0-5][0-9]):([0-5][0-9])(\.[0-9]+)?": "
(Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Regex-Optionen: Keine

Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[0-1][0-9]):(?<minute>[0-5][0-9]):(?<second>[0-5][0-9])": "
(?<ms>\.[0-9]+)?(?<timezone>Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Regex-Optionen: Keine

Regex-Varianten: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Datum und Uhrzeit mit optionalen Sekundenbruchteilen und Zeitzone, zum Beispiel `2008-08-30T01:45:36` oder `2008-08-30T01:45:36.123Z`. Das ist der XML Schema-Typ `dateTime`:

```
^(-?(?:[1-9][0-9]*)?[0-9]{4})-(1[0-2]|0[1-9])-(3[0-1]|0[1-9]|[1-2][0-9])": "
T(2[0-3]|[0-1][0-9]):([0-5][0-9]):([0-5][0-9])(\.[0-9]+)?": "
(Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Regex-Optionen: Keine

Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>-?(?:[1-9][0-9]*)?[0-9]{4})-(?<month>1[0-2]|0[1-9])-": "
(?<day>3[0-1]|0[1-9]|[1-2][0-9])T(?<hour>2[0-3]|[0-1][0-9]):": "
(?<minute>[0-5][0-9]):(?<second>[0-5][0-9])(?<ms>\.[0-9]+)?": "
(?<timezone>Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Regex-Optionen: Keine

Regex-Varianten: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Diskussion

Die ISO 8601 definiert eine große Anzahl von Datums- und Zeitformaten. Die hier vorgestellten regulären Ausdrücke kümmern sich um die gebräuchlichsten Formate, aber die meisten Systeme, die ISO 8601 verwenden, greifen nur auf eine Untermenge zurück. So sind zum Beispiel bei Datums- und Zeitwerten in XML Schema die Bindestriche und

Doppelpunkte nicht optional. Um diese Zeichen in den regulären Ausdrücken einzufordern, entfernen Sie einfach die Fragezeichen hinter ihnen. Passen Sie aber auf nicht-einfangende Gruppen auf, die die Syntax ‹(?:Gruppe)› verwenden. Wenn ein Fragezeichen und ein Doppelpunkt auf eine öffnende Klammer folgen, sind diese drei Zeichen der Anfang einer nicht-einfangenden Gruppe.

Bei den regulären Ausdrücken sind die einzelnen Bindestriche und Doppelpunkte optional, was nicht ganz ISO 8601 entspricht. So ist zum Beispiel 1733:26 nach ISO 8601 keine gültige Zeit, sie wird aber von den Zeit-Regexes akzeptiert. Wenn alle Bindestriche und Doppelpunkte gleichzeitig vorhanden oder eben nicht vorhanden sein sollen, wird Ihre Regex ein ganzes Stück komplexer. Wir haben das als Beispiel für die Datums-Regex gezeigt, aber im Alltag sind die Trennzeichen im Allgemeinen entweder auf jeden Fall erforderlich (wie bei XML Schema) oder immer verboten und nicht optional.

Wir haben um alle Zahlenelemente der Regex Klammern gelegt. Damit ist es einfach, die Werte für Jahr, Monat, Tag, Stunden, Minuten, Sekunden und Zeitzonen auszulesen. In Rezept 2.9 wird beschrieben, wie Klammern einfangende Gruppen definieren. Rezept 3.9 zeigt Ihnen, wie Sie den von diesen einfangenden Gruppen gefundenen Text mithilfe von prozeduralem Code auslesen können.

Bei den meisten Regexes haben wir auch eine Alternative mit benannten Captures präsentiert. Manche dieser Datums- und Zeitformate sind Ihnen oder Ihren Kollegen vielleicht unbekannt. Benannte Captures erleichtern das Verstehen der Regex. .NET, PCRE 7, Perl 5.10 und Ruby 1.9 unterstützen die Syntax ‹(?‹Name›Gruppe)›. Alle Versionen von PCRE und Python, die in diesem Buch behandelt werden, bieten auch die alternative Syntax ‹(?P‹Name›Gruppe)› an, in der ein zusätzliches ‹P› enthalten ist. In Rezept 2.11 und Rezept 3.9 finden Sie die Details dazu.

Die Zahlenbereiche in allen Regexes sind sehr strikt. So ist zum Beispiel der Kalendertag auf Werte zwischen 01 und 31 eingeschränkt. Sie werden nie einen Tag 32 oder einen Monat 13 erhalten. Allerdings versucht keine der vorgestellten Regexes, ungültige Kombinationen aus Tag und Monat auszuschließen, wie zum Beispiel den 31. Februar. In Rezept 4.5 beschreiben wir, wie Sie dieses Problem angehen können.

Auch wenn manche dieser Regexes ziemlich lang sind, gibt es in ihnen dennoch keine exotischen Verrenkungen, und alle nutzen die gleichen Techniken, die in Rezept 4.4 und Rezept 4.6 erläutert wurden.

Siehe auch

Rezepte 4.4, 4.5, 4.6.

4.8 Eingabe auf alphanumerische Zeichen beschränken

Problem

Bei Ihrer Anwendung soll der Nutzer bei einer Eingabe nur alphanumerische Zeichen aus dem englischen Alphabet eingeben dürfen.

Lösung

Mit den Ihnen zur Verfügung stehenden regulären Ausdrücken ist die Lösung ganz einfach. Eine Zeichenklasse kann den erlaubten Bereich mit Zeichen festlegen. Mit einem Quantor, der die Zeichenklasse ein Mal oder mehrfach zulässt, und Ankern, die die Übereinstimmung mit dem Anfang und dem Ende des Strings verbinden, sind Sie schon fertig.

Regulärer Ausdruck

```
^[A-Z0-9]+$
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ruby

```
if subject =~ /^[A-Z0-9]+$/i
    puts "Text ist alphanumerisch"
else
    puts "Text ist nicht alphanumerisch"
end
```

Andere Programmiersprachen

In den Rezepten 3.4 und 3.5 finden Sie Informationen über das Implementieren dieses regulären Ausdrucks in anderen Programmiersprachen.

Diskussion

Lassen Sie uns die vier Teile dieses regulären Ausdrucks nacheinander anschauen:

```
^         # Sicherstellen, dass die Übereinstimmung am Anfang des Texts beginnt.
[A-Z0-9]  # Ein Zeichen zwischen "A" und "Z" oder zwischen "0" und "9" finden...
  +       #   einmal bis unbegrenzt oft.
$         # Sicherstellen, dass die Übereinstimmung am Ende des Texts endet.
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert, Freiform-Modus
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Die Anker ‹^› und ‹$› am Anfang und Ende des regulären Ausdrucks sorgen dafür, dass die gesamte Eingabe überprüft wird. Ohne sie könnte die Regex auch Teile eines längeren Strings finden, sodass doch ungültige Zeichen übersehen werden. Durch den Quantor ‹+› kann das vorherige Element einmal oder häufiger vorkommen. Wenn Ihre Regex auch einen vollständig leeren String erkennen soll, können Sie das ‹+› durch ‹*› ersetzen. Der Stern-Quantor ‹*› erlaubt, dass ein Element null Mal oder häufiger vorkommt, wodurch dieses Element im Endeffekt optional wird.

Variationen

Eingabe auf ASCII-Zeichen einschränken

Der folgende reguläre Ausdruck beschränkt die Eingabe auf die 128 Zeichen in der 7-Bit-ASCII-Tabelle. Dazu gehören auch 33 nicht sichtbare Steuerzeichen:

```
^[\x00-\x7F]+$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Eingabe auf ASCII-Zeichen ohne Steuerzeichen und auf Zeilenumbrüche einschränken

Mit dem folgenden regulären Ausdruck beschränken Sie die Eingabe auf sichtbare Zeichen und Whitespace in der ASCII-Tabelle. Steuerzeichen werden damit ausgeschlossen. Die Zeichen für Line Feed und Carriage Return (mit den Werten 0x0A und 0x0D) sind die gebräuchlichsten Steuerzeichen, daher werden sie hier explizit durch ‹\n› (Line Feed) und ‹\r› (Carriage Return) mit aufgenommen:

```
^[\n\r\x20-\x7E]+$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Eingabe auf die Zeichen begrenzen, die sowohl in ISO-8859-1 als auch in Windows-1252 vorkommen

ISO-8859-1 und Windows-1252 (häufig als ANSI bezeichnet) sind zwei häufig genutzte Zeichenkodierungen mit 8 Bit Breite, die beide auf dem Latin-1-Standard basieren (genauer gesagt, auf ISO/IEC 8859-1). Allerdings sind die Zeichen an den Positionen von 0x80 und 0x9F inkompatibel. ISO-8859-1 nutzt diese Positionen für Steuerzeichen, während Windows-1252 dort noch mehr Buchstaben und Satzzeichen abgelegt hat. Diese Unterschiede führen manchmal zu Problemen beim Anzeigen von Zeichen, insbesondere bei Dokumenten, die ihre Kodierung nicht angeben, oder bei denen der Empfänger ein Nicht-Windows-System verwendet. Der folgende reguläre Ausdruck kann dazu verwendet werden, die Eingabe auf Zeichen zu beschränken, die in beiden Zeichentabellen ISO-8859-1 und Windows-1252 vorhanden sind (einschließlich der gemeinsamen Steuerzeichen):

```
^[\x00-\x7F\xA0-\xFF]+$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Durch die hexadezimale Schreibweise ist dieser reguläre Ausdruck vielleicht etwas schlechter zu lesen, aber er arbeitet genau so wie die weiter oben gezeigte Zeichenklasse ‹[A-Z0-9]›. Die Regex passt auf Zeichen in zwei Bereichen: \x00-\x7F und \xA0-\xFF.

Eingabe auf alphanumerische Zeichen in beliebigen Sprachen einschränken

Dieser reguläre Ausdruck beschränkt die Eingabe auf Buchstaben und Ziffern aus beliebigen Sprachen oder Schriften. Er verwendet eine Zeichenklasse, die Eigenschaften für alle Codepoints in den Buchstaben- und Ziffernkategorien von Unicode enthält:

```
^[\p{L}\p{N}]+$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, PCRE, Perl, Ruby 1.9

Leider werden Unicode-Eigenschaften nicht von allen in diesem Buch behandelten Regex-Varianten unterstützt. Vor allem funktioniert diese Regex nicht in JavaScript, Python und Ruby 1.8. Zudem muss die PCRE mit UTF-8-Unterstützung kompiliert werden, will man diese Regex dort nutzen. Unicode-Eigenschaften können in den preg-Funktionen von PHP genutzt werden (die auf der PCRE aufbauen), wenn an die Regex die Option /u angehängt wird.

Mit der folgenden Regex kann man in Python die fehlende Unterstützung von Unicode-Eigenschaften umgehen:

```
^[^\W_]+$
```

Regex-Optionen: Unicode
Regex-Variante: Python

Hier umgehen wir die in Python fehlenden Unicode-Eigenschaften, indem wir den Schalter UNICODE oder U beim Erstellen des regulären Ausdrucks verwenden. Dadurch wird die Bedeutung einiger Regex-Tokens so verändert, dass sie auf die Unicode-Zeichentabelle zugreifen. ‹\w› bringt uns schon recht weit, weil damit alphanumerische Zeichen und der Unterstrich gefunden werden. Durch die inverse Abkürzung ‹\W› in einer negierten Zeichenklasse können wir den Unterstrich aus dieser Menge entfernen. Solche doppelt negierten Elemente sind in regulären Ausdrücken manchmal recht nützlich, auch wenn man eventuell erst einmal seinen Grips dafür anstrengen muss.[1]

1 Wenn Sie noch mehr Spaß haben wollen (für sehr seltsame Definitionen von „Spaß"), können Sie versuchen, dreifache, vierfache oder noch „höher-fache" Negierungen zu erzeugen, indem Sie negative Lookarounds (siehe Rezept 2.16) und Zeichenklassen-Subtraktionen (siehe Rezept 2.3) ins Spiel bringen.

Siehe auch

In Rezept 4.9 wird gezeigt, wie man die Länge des Texts einschränkt.

4.9 Die Länge des Texts begrenzen

Problem

Sie wollen prüfen, ob ein String zwischen einem und zehn Buchstaben von A bis Z enthält.

Lösung

Alle Programmiersprachen, die in diesem Buch behandelt werden, stellen eine einfache und effiziente Möglichkeit bereit, die Länge eines Texts zu überprüfen. So haben zum Beispiel JavaScript-Strings eine Eigenschaft `length` mit einer Integer-Zahl, die die Länge des Strings angibt. Aber manchmal kann es auch sinnvoll sein, die Länge eines Texts mit einem regulären Ausdruck zu prüfen, insbesondere wenn die Länge nur eine von mehreren Regeln ist, die bestimmen, ob der Ausgangstext in das gewünschte Muster passt. Der folgende reguläre Ausdruck stellt sicher, dass der Text zwischen 1 und 10 Zeichen lang ist. Zudem schränkt er den Text auf die Großbuchstaben A bis Z ein. Sie können die regulären Ausdrücke so verändern, dass sie eine minimale oder maximale Textlänge definieren, aber auch andere Zeichen als A bis Z zulassen.

Regulärer Ausdruck

```
^[A-Z]{1,10}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Perl

```
if ($ARGV[0] =~ /^[A-Z]{1,10}$/) {
    print "Eingabe ist gültig\n";
} else {
    print "Eingabe ist ungültig\n";
}
```

Andere Programmiersprachen

In Rezept 3.5 finden Sie Informationen darüber, wie Sie diesen regulären Ausdruck in anderen Programmiersprachen implementieren können.

Diskussion

Hier ist die Aufteilung dieser sehr einfachen Regex in ihre Bestandteile:

```
^         # Sicherstellen, dass die Regex am Textanfang passt.
[A-Z]     # Einen der Buchstaben von "A" bis "Z" finden ...
  {1,10}  #   zwischen 1 und 10 Mal.
$         # Sicherstellen, dass die Regex am Textende passt.
```

Regex-Optionen: Freiform
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Die Anker ‹^› und ‹$› stellen sicher, dass die Regex den gesamten Ausgangstext umfasst. Ansonsten könnte es sein, dass sie nur zehn Zeichen innerhalb eines längeren Texts findet. Die Zeichenklasse ‹[A-Z]› passt auf einen einzelnen Großbuchstaben von A bis Z. Der Intervall-Quantor ‹{1,10}› sorgt dafür, dass die Zeichenklasse zwischen einem und zehn Mal vorkommen kann. Indem man den Intervall-Quantor mit den Textanfangs- und -ende-Ankern kombiniert, wird die Regex fehlschlagen, wenn die Länge des Ausgangstexts außerhalb des gewünschten Bereichs liegt.

Beachten Sie, dass die Zeichenklasse ‹[A-Z]› explizit nur Großbuchstaben zulässt. Wenn Sie auch die Kleinbuchstaben a bis z aufnehmen wollen, können Sie entweder als Zeichenklasse ‹[A-Za-z]› verwenden oder die Option zum Ignorieren von Groß- und Kleinschreibung nutzen. In Rezept 3.4 ist nachzulesen, wie man das tut.

Ein von Anfängern häufig gemachter Fehler ist, ein paar Tastendrücke dadurch zu sparen, dass man den Zeichenklassenbereich ‹[A-z]› nutzt. Auf den ersten Blick sieht das eigentlich ganz praktisch aus. Aber die ASCII-Zeichentabelle enthält zwischen den Bereichen von A bis Z und a bis z eine Reihe von Satzzeichen. Daher entspricht ‹[A-z]› in Wirklichkeit ‹[A-Z[\]^_`a-z]›.

Variationen

Die Länge eines bestimmten Musters begrenzen

Da Quantoren wie ‹{1,10}› nur auf das direkt vor ihnen stehende Element wirken, muss man etwas anders vorgehen, wenn man die Zeichenzahl von Mustern begrenzen will, die mehr als ein einzelnes Token enthalten.

Wie in Rezept 2.16 beschrieben, sind Lookaheads (und ihre Gegenstücke, die Lookbehinds) besondere Arten von Zusicherungen, die wie ‹^› und ‹$› auf eine Position innerhalb des Ausgangstexts passen, aber keine Zeichen konsumieren. Lookaheads können entweder positiv oder negativ sein. Das bedeutet, man kann mit ihnen prüfen, ob auf die aktuelle Position ein Muster folgt – oder eben nicht. Ein positives Lookahead, das die Syntax ‹(?=···)› hat, kann am Anfang des Musters genutzt werden, um sicherzustellen, dass die Länge des Strings innerhalb des Zielbereichs liegt. Der Rest der Regex kann dann das gewünschte Muster prüfen, ohne sich darum kümmern zu müssen, wie lang der Text ist. Hier ein einfaches Beispiel:

```
^(?=.{1,10}$).*
```

Regex-Optionen: Punkt passt zu Zeilenumbruch
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?=[\S\s]{1,10}$)[\S\s]*
```

Regex-Optionen: Keine
Regex-Variante: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Es ist wichtig, dass sich der Anker ‹$› innerhalb des Lookaheads befindet, denn das Prüfen der maximalen Länge funktioniert nur, wenn wir sicherstellen, dass es nach dem Erreichen der Grenze keine weiteren Zeichen gibt. Da das Lookahead die Länge des Bereichs am Anfang der Regex sicherstellt, kann das folgende Muster dann beliebige zusätzliche Validierungsregeln umsetzen. In diesem Fall wird das Muster ‹.*› (oder ‹[\S\s]*› für JavaScript) genutzt, um einfach den gesamten Ausgangstext zu finden – ohne zusätzliche Einschränkungen.

Die erste Regex verwendet die Option *Punkt passt auf Zeilenumbruch*, damit der Punkt wirklich alle Zeichen findet – auch Zeilenumbrüche. In Rezept 3.4 finden Sie Details über das Anwenden dieses Modifikators in Ihrer Programmiersprache. Die Regex für JavaScript sieht anders aus, da JavaScript die Option *Punkt passt auf Zeilenumbruch* nicht besitzt. In „Jedes Zeichen einschließlich Zeilenumbrüchen“ auf Seite 37 in Rezept 2.4 finden Sie weitere Informationen.

Anzahl der Zeichen ohne Whitespace einschränken

Die folgende Regex passt auf Strings, die zwischen 10 und 100 Nicht-Whitespace-Zeichen enthalten:

```
^\s*(?:\S\s*){10,100}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

In Java, PCRE, Python und Ruby passt ‹\s› nur auf ASCII-Whitespace-Zeichen und ‹\S› auf alles andere. In Python können Sie durch die Option `UNICODE` oder `U` beim Erzeugen der Regex dafür sorgen, dass ‹\s› jeden Unicode-Whitespace berücksichtigt. Entwickler, die mit Java, der PCRE oder Ruby 1.9 arbeiten und verhindern wollen, dass Unicode-Whitespace beim Zählen berücksichtigt wird, können die folgende Version nutzen, die von den Unicode-Eigenschaften Gebrauch macht (beschrieben in Rezept 2.7):

```
^[\p{Z}\s]*(?:[^\p{Z}\s][\p{Z}\s]*){10,100}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, PCRE, Perl, Ruby 1.9

Die PCRE muss mit UTF-8-Unterstützung kompiliert werden, damit diese Regex wie gewünscht funktioniert. In PHP müssen Sie die UTF-8-Unterstützung durch den Modifikator `/u` aktivieren.

Diese Regex kombiniert die Unicode-Eigenschaft „Separator“ ‹\p{Z}› mit der Zeichenklassenabkürzung ‹\s› für Whitespace. Das liegt daran, dass Zeichen, die von ‹\p{Z}› und ‹\s› gefunden werden, nicht unbedingt identisch sind. Zu ‹\s› gehören die Zeichen an den Positionen 0x09 bis 0x0D (Tab, Line Feed, vertikaler Tab, Form Feed und Carriage Return), die nicht der Separator-Eigenschaft im Unicode-Standard zugewiesen sind. Indem Sie ‹\p{Z}› und ‹\s› in einer Zeichenklasse kombinieren, stellen Sie sicher, dass alle Whitespace-Zeichen gefunden werden.

In beiden Regexes wird der Intervall-Quantor ‹{10,100}› auf die vor ihm stehende nichteinfangende Gruppe angewendet und nicht nur auf ein einzelnes Token. Die Gruppe passt zu jedem einzelnen Nicht-Whitespace-Zeichen, dem null oder mehr Whitespace-Zeichen folgen. Der Intervall-Quantor kann zuverlässig erkennen, wie viele Nicht-Whitespace-Zeichen gefunden wurden, da während jeder Iteration nur genau ein Nicht-Whitespace-Zeichen passt.

Anzahl der Wörter einschränken

Die folgende Regex ähnelt dem vorigen Beispiel, bei dem die Anzahl der Nicht-Whitespace-Zeichen begrenzt wird. Nur passt hier jede Wiederholung auf ein ganzes Wort und nicht auf ein einzelnes Nicht-Whitespace-Zeichen. Es passt auf 10 bis 100 Wörter, wobei alle Nicht-Wortzeichen ignoriert werden – auch Satzzeichen und Whitespace:

```
^\W*(?:\w+\b\W*){10,100}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

In Java, JavaScript, PCRE und Ruby passt das Wortzeichen-Token ‹\w› in dieser Regex nur auf die ASCII-Zeichen A–Z, a–z, 0–9 und _. Daher lassen sich damit Wörter mit Nicht-ASCII-Buchstaben und -Zahlen nicht korrekt zählen. In .NET und Perl basiert ‹\w› auf der Unicode-Tabelle (genauso wie das Gegen-Token ‹\W› und die Wortgrenze ‹\b›) und passt daher auf Buchstaben und Ziffern aus allen Unicode-Schriftsystemen. In Python können Sie selbst wählen, ob diese Tokens auf Unicode basieren sollen oder nicht. Das hängt davon ob, ob Sie die Option `UNICODE` oder `U` beim Erstellen der Regex mit angeben.

Wenn Sie Wörter zählen wollen, die Nicht-ASCII-Buchstaben und -Zahlen enthalten, können die folgenden Regexes dies für weitere Regex-Varianten ermöglichen:

```
^[^\p{L}\p{N}_]*(?:[\p{L}\p{N}_]+\b[^\p{L}\p{N}_]*){10,100}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, Perl

```
^[^\p{L}\p{N}_]*(?:[\p{L}\p{N}_]+(?:[^\p{L}\p{N}_]+|$)){10,100}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, PCRE, Perl, Ruby 1.9

Die PCRE muss mit UTF-8-Unterstützung kompiliert werden, damit diese Regex dort funktioniert. In PHP schalten Sie die UTF-8-Unterstützung mit dem Muster-Modifikator `/u` ein.

Wie schon in „Wortzeichen“ auf Seite 46 in Rezept 2.6 erwähnt, ist der Grund für diese verschiedenen (aber gleich funktionierenden) Regexes der unterschiedliche Umgang mit Wortzeichen und Wortgrenzen.

Die letzten beiden Regexes nutzen Zeichenklassen, die die Unicode-Eigenschaften für Buchstaben und Zahlen verwenden (`‹\p{L}›` und `‹\p{N}›`). Dazu wurde noch der Unterstrich mit aufgenommen, damit sich die Zeichenklassen genau so verhalten wie bei den anderen Regexes, die auf `‹\w›` und `‹\W›` aufbauen.

Jede Wiederholung der nicht-einfangenden Gruppe in den ersten beiden dieser drei Regexes passt zu einem vollständigen Wort, auf das null oder mehr Nicht-Wortzeichen folgen. Das Token `‹\W›` (oder `‹[^\p{L}\p{N}_]›`) innerhalb der Gruppe ist optional, falls der String mit einem Wortzeichen endet. Da damit aber die Nicht-Wortzeichen-Folge während des Suchprozesses optional würde, brauchen wir die Wortgrenzenzusicherung `‹\b›` zwischen `‹\w›` und `‹\W›` (oder `‹[\p{L}\p{N}_]›` und `‹[^\p{L}\p{N}_]›`). Damit ist sichergestellt, dass jede Wiederholung der Gruppe wirklich ein vollständiges Wort findet. Ohne die Wortgrenze würde eine einzelne Wiederholung einen beliebigen Teil eines Worts finden, und die folgenden Wiederholungen könnten dann auf zusätzliche Teile passen.

Die dritte Version der Regex (durch die auch PCRE und Ruby 1.9 mitmachen können) funktioniert ein bisschen anders. Sie verwendet einen Plus- (eins oder mehr) statt eines Stern-Quantors (null oder mehr) und lässt explizit nur dann null Zeichen zu, wenn der Suchprozess schon am Ende des Strings angelangt ist. Damit wird das Wortgrenzen-Token vermieden, was für eine genauere Suche wichtig war, da `‹\b›` in der PCRE und in Ruby nicht Unicode-kompatibel ist. In Java ist `‹\b›` Unicode-kompatibel, `‹\w›` allerdings nicht.

Leider ist es mit keiner dieser Optionen möglich, in JavaScript oder Ruby 1.8 korrekt mit Wörtern umzugehen, die Nicht-ASCII-Zeichen verwenden. Eine mögliche Alternative ist, die Regex so umzubauen, dass sie Whitespace-Sequenzen statt Wörter zählt:

```
^\s*(?:\S+(?:\s+|$)){10,100}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, Perl, PCRE, Python, Ruby

Das funktioniert in vielen Fällen genau so wie die vorherigen Lösungen, ist aber nicht genau das Gleiche. So werden hier zum Beispiel Wörter mit Bindestrichen (wie „S-Bahn“) als ein Wort gezählt und nicht mehr als zwei Wörter.

Siehe auch

Rezepte 4.8 und 4.10.

4.10 Die Zeilenanzahl eines Texts beschränken

Problem

Sie müssen prüfen, ob ein String aus fünf oder weniger Zeilen besteht, wobei die Gesamtlänge des Strings unwichtig ist.

Lösung

Die Zeichen oder Zeichenfolgen, die als Zeilentrenner genutzt werden, können sehr stark von Ihrem Betriebssystem, der Anwendung oder den Einstellungen des Benutzers abhängig sein. Daher stellt sich beim Schaffen einer idealen Lösung die Frage, welche Konventionen unterstützt werden sollen, um den Anfang einer neuen Zeile zu erkennen. Die folgenden Lösungen unterstützen den Standard von MS-DOS/Windows (‹\r\n›), dem alten Mac OS (‹\r›) und Unix/Linux/OS X (‹\n›).

Regulärer Ausdruck

Die folgenden drei variantenspezifischen Regexes besitzen zwei Unterschiede. Die erste Regex verwendet atomare Gruppen, die als ‹(?>···)› geschrieben werden, statt auf nicht-einfangende Gruppen zurückzugreifen (‹(?:···)›), denn dadurch können die Regex-Varianten, die sie unterstützen, eventuell einen kleinen Geschwindigkeitsvorteil erhalten. Python und JavaScript bieten keine atomaren Gruppen, daher werden sie bei diesen Varianten nicht verwendet. Der andere Unterschied sind die Tokens, die genutzt werden, um die Position am Anfang und Ende des Strings sicherzustellen (‹\A› oder ‹^› für den Anfang des Strings und ‹\z›, ‹\Z› oder ‹$› für dessen Ende). Die Gründe für diese Unterschiede werden später noch genauer beleuchtet. Alle drei variantenspezifischen Regexes passen auf genau die gleichen Strings:

```
\A(?>(?>\r\n?|\n)?[^\r\n]*){0,5}\z
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, PCRE, Perl, Ruby

```
\A(?:(?:\r\n?|\n)?[^\r\n]*){0,5}\Z
```

Regex-Optionen: Keine
Regex-Variante: Python

```
^(?:(?:\r\n?|\n)?[^\r\n]*){0,5}$
```

Regex-Optionen: Keine
Regex-Variante: JavaScript

PHP (PCRE)

```
if (preg_match('/\A(?>(?>\r\n?|\n)?[^\r\n]*){0,5}\z/', $_POST['subject'])) {
    print 'Text enthält fünf oder weniger Zeilen';
} else {
    print 'Text enthält mehr als fünf Zeilen';
}
```

Andere Programmiersprachen

In Rezept 3.5 finden Sie Informationen dazu, wie diese regulären Ausdrücke mit anderen Programmiersprachen implementiert werden können.

Diskussion

Alle in diesem Rezept bis hierhin gezeigten regulären Ausdrücke nutzen eine Gruppe, die zu einem Zeilenumbruch in MS-DOS/Windows, dem alten Mac OS oder in Unix/Linux/OS X passen, gefolgt von einer beliebigen Zahl von Zeichen, die kein Zeilenumbruch sind. Diese Gruppe wird zwischen null und fünf Mal wiederholt, da wir bis zu fünf Zeilen finden wollen.

Im folgenden Beispiel haben wir die JavaScript-Version der Regex in ihre Bestandteile zerlegt. Die JavaScript-Version haben wir hier deshalb verwendet, weil ihre Elemente vermutlich den meisten Lesern bekannt sein dürften. Wir werden die Versionen für andere Regex-Varianten im Anschluss behandeln:

```
^          # Position am Anfang des Strings sicherstellen.
(?:        # Gruppieren, aber nicht einfangen ...
  (?:      #   Gruppieren, aber nicht einfangen ...
    \r     #     Ein Carriage Return (CR, ASCII-Position 0x0D) finden.
    \n     #     Ein Line Feed (LF, ASCII-Position 0x0A) finden ...
      ?    #       null oder ein Mal.
   |       #    oder ...
    \n     #     Ein Line Feed finden.
  )        #   Ende der nicht-einfangenden Gruppe.
    ?      #     Die vorige Gruppe null oder ein Mal wiederholen.
  [^\r\n]  #   Ein beliebiges Zeichen außer CR oder LF finden ...
    *      #     null Mal oder öfter.
)          # Ende der nicht-einfangenden Gruppe.
  {0,5}    #   Vorherige Gruppe null bis fünf Mal wiederholen.
$          # Position am Ende des Strings sicherstellen.
```

Regex-Optionen: Freiform

Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Das führende ‹^› stellt sicher, dass sich die Übereinstimmung am Anfang des Strings befindet. Damit wird dafür gesorgt, dass der gesamte String nicht mehr als fünf Zeilen enthält, denn so können sich nicht schon vor dem gefundenen Bereich Zeilen befinden.

Als Nächstes umschließt eine nicht-einfangende Gruppe die Kombination einer Zeilenumbruchfolge und einer beliebigen Zahl von Zeichen, die gerade kein Zeilenumbruch

sind. Der direkt darauffolgende Quantor erlaubt, diese Gruppe zwischen null und fünf Mal zu finden (null Wiederholungen passen zu einem vollständig leeren String). Innerhalb der äußeren Gruppe passt eine optionale Untergruppe zu einer Zeilenumbruchfolge. Danach kommt die Zeichenklasse, die zu einer beliebigen Zahl von Zeichen passt, die kein Zeilenumbruch sind.

Schauen Sie sich die Reihenfolge der Elemente der äußeren Gruppe genauer an (zuerst ein Zeilenumbruch, dann anderer Text). Wenn wir die Reihenfolge umkehren, sodass die Gruppe stattdessen als ‹(?:[^\r\n]*(?:\r\n?|\n)?)› geschrieben würde, ließe eine fünfte Wiederholung einen abschließenden Zeilenumbruch zu. Somit würde man eine leere sechste Zeile zulassen.

Die Untergruppe erlaubt eine von drei Zeilenumbruchfolgen:

- Ein Carriage Return, gefolgt von einem Line Feed (‹\r\n›, der normale Zeilenumbruch im MS-DOS-/Windows-Umfeld).
- Ein einzelnes Carriage Return (‹\r›, der Zeilenumbruch im alten Mac OS).
- Ein einzelnes Line Feed (‹\n›, der klassische Zeilenumbruch unter Unix/Linux/OS X).

Lassen Sie uns nun die Unterschiede bei den Varianten anschauen.

Die erste Version der Regex (die für alle Varianten außer Python und JavaScript genutzt werden kann) verwendet atomare Gruppen statt einfache nicht-einfangende Gruppen. Auch wenn die Verwendung von atomaren Gruppen einen deutlich größeren Einfluss auf die Performance haben kann, wird die Regex-Engine in diesem Fall nur vor ein bisschen unnötigem Backtracking bewahrt, das bei einer fehlschlagenden Suche auftreten kann (in Rezept 2.15 erhalten Sie mehr Informationen über atomare Gruppen).

Die anderen Unterschiede zwischen den Varianten sind die Tokens, die genutzt werden, um die Position am Anfang und Ende des Strings sicherzustellen. Die auseinandergenommene Regex weiter oben hat dafür ‹^› und ‹$› genutzt. Diese Anker werden zwar von allen hier behandelten Regex-Varianten unterstützt, die anderen Regexes in diesem Abschnitt nutzen aber stattdessen ‹\A›, ‹\Z› und ‹\z›. Kurz gesagt, unterscheidet sich die Bedeutung dieser Metazeichen ein wenig bei den verschiedenen Regex-Varianten. Eine ausführlichere Erklärung lässt uns ein wenig in die Geschichte von Regexes eintauchen ...

Wenn man Perl nutzt, um eine Zeile aus einer Datei einzulesen, endet der sich so ergebende String mit einem Zeilenumbruch. Daher hat Perl eine „Verbesserung“ für die klassische Bedeutung von ‹$› eingeführt, die seitdem von den meisten Regex-Varianten übernommen wurde. So findet ‹$› nicht nur das absolute Ende des Strings, sondern auch einen Zeilenumbruch direkt vor dem String-Ende. Perl hat zudem zwei weitere Zusicherungen für das Ende eines Strings eingeführt: ‹\Z› und ‹\z›. Der Anker ‹\Z› hat die gleiche eigenartige Bedeutung wie ‹$›, nur dass sich diese nicht ändert, wenn die Option *Zirkumflex und Dollar passen auf Zeilenumbruch* für ‹^› und ‹$› eingeschaltet wird. ‹\z› passt immer nur auf das absolute Ende eines Strings – ohne Ausnahme. Da dieses Rezept explizit mit Zeilenumbrüchen hantiert, um die Zeilen in einem String zu zählen, nutzt es

die Zusicherung ‹\z› für die Regex-Varianten, in denen sie angeboten wird. Damit kann sichergestellt werden, dass es keine sechste, leere Zeile gibt.

Die meisten anderen Regex-Varianten haben die Zeilenende-/String-Ende-Anker von Perl übernommen. .NET, Java, PCRE und Ruby unterstützen alle sowohl ‹\Z› als auch ‹\z› mit der gleichen Bedeutung wie in Perl. Python bietet nur das ‹\Z› (als Großbuchstabe), wobei es aber verwirrenderweise die Bedeutung ändert. Es passt nur auf das absolute Ende des Strings, so wie das kleine ‹\z› von Perl. JavaScript unterstützt überhaupt keine „z"-Anker, aber anders als die anderen Varianten passt sein Anker ‹$› nur auf das absolute Ende des Strings (wenn die Option *Zirkumflex und Dollar passen auf Zeilenumbruch* nicht aktiv ist).

Bei ‹\A› ist das Ganze etwas übersichtlicher. Dieser Anker passt immer nur auf den Anfang eines Strings und hat überall die gleiche Bedeutung – außer in JavaScript, das ihn nicht unterstützt.

Es ist zwar nicht sehr schön, dass es diese verwirrenden Inkonsistenzen gibt, aber einer der Vorteile beim Verwenden regulärer Ausdrücke in diesem Buch ist, dass Sie sich normalerweise keine Sorgen darum machen müssen. Solche unschönen Details werden nur dann relevant, wenn Sie sich doch intensiver mit den Regexes befassen wollen.

Variationen

Umgang mit esoterischen Zeilentrennern

Die oben gezeigten Regexes unterstützen nur die klassischen Zeilenumbrüche von MS-DOS/Windows, Unix/Linux/OS X und dem alten Mac OS. Aber es gibt noch eine Reihe selten genutzter vertikaler Whitespace-Zeichen, die Ihnen gelegentlich über den Weg laufen. Die folgenden Regexes berücksichtigen diese zusätzlichen Zeichen, und schränken dabei die Übereinstimmungen auf maximal fünf Zeilen Text ein.

```
\A(?>\R?\V*){0,5}\z
```

Regex-Optionen: Keine
Regex-Varianten: PCRE 7 (mit der Option PCRE_BSR_UNICODE), Perl 5.10

```
\A(?>(?>\r\n?|[\n-\f\x85\x{2028}\x{2029}])?: "
[^\n-\r\x85\x{2028}\x{2029}]*){0,5}\z
```

Regex-Optionen: Keine
Regex-Varianten: PCRE, Perl

```
\A(?>(?>\r\n?|[\n-\f\x85\u2028\u2029])?[^\n-\r\x85\u2028\u2029]*){0,5}\z
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, Ruby

```
\A(?:(?:\r\n?|[\n-\f\x85\u2028\u2029])?[^\n-\r\x85\u2028\u2029]*){0,5}\Z
```

Regex-Optionen: Keine
Regex-Variante: Python

```
^(?:(?:\r\n?|[\n-\f\x85\u2028\u2029])?[^\n-\r\x85\u2028\u2029]*){0,5}$
```

Regex-Optionen: Keine
Regex-Variante: JavaScript

Alle diese Regexes kümmern sich um die Zeilentrenner aus Tabelle 4-1, die zusammen mit ihren Unicode-Positionen und Namen aufgeführt sind.

Tabelle 4-1: Zeilentrenner

Unicode-Folge	Regex-Äquivalent	Name	Verwendung
U+000D U+000A	‹\r\n›	Carriage Return und Line Feed (CRLF)	Textdateien unter Windows und MS-DOS
U+000A	‹\n›	Line Feed (LF)	Textdateien unter Unix, Linux und OS X
U+000B	‹\v›	Line Tabulation (auch vertikaler Tab oder VT)	(selten)
U+000C	‹\f›	Form Feed (FF)	(selten)
U+000D	‹\r›	Carriage Return (CR)	Textdateien unter Mac OS
U+0085	‹\x85›	Next Line (NEL)	Textdateien auf IBM Mainframes (selten)
U+2028	‹\u2028› oder ‹\x{2028}›	Zeilentrenner (Line Separator)	(selten)
U+2029	‹\u2029› oder ‹\x{2029}›	Absatztrenner (Paragraph Separator)	(selten)

Siehe auch

Rezept 4.9.

4.11 Antworten auswerten

Problem

Sie müssen eine Konfigurationsoption oder eine Eingabe an der Befehlszeile auf einen positiven Wert überprüfen. Sie wollen bei den möglichen Antworten flexibel sein, sodass `true`, `t`, `yes`, `y`, `ja`, `j`, `okay`, `ok` und `1` in beliebiger Groß- und Kleinschreibung akzeptiert werden.

Lösung

Mit einer Regex, die alle akzeptablen Antworten kombiniert, können Sie die Überprüfung mit einem einfachen Test durchführen.

Regulärer Ausdruck

```
^(?:1|t(?:rue)?|y(?:es)?|ja?|ok(?:ay)?)$
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

JavaScript

```
var yes = /^(?:1|t(?:rue)?|y(?:es)?|ja?|ok(?:ay)?)$/i;

if (yes.test(subject)) {
    alert("Ja");
} else {
    alert("Nein");
}
```

Andere Programmiersprachen

In den Rezepten 3.4 und 3.5 erhalten Sie Hinweise dazu, wie dieser reguläre Ausdruck in anderen Programmiersprachen implementiert werden kann.

Diskussion

Die folgende Regex zeigt die einzelnen Elemente im Detail. Kombinationen von Tokens, die leicht zusammen lesbar sind, werden in einer Zeile aufgeführt:

```
^            # Position am Anfang des Strings sicherstellen.
(?:          # Gruppieren, aber nicht einfangen ...
  1          #   Eine literale "1" finden.
 |           #  oder ...
  t(?:rue)?  #   Finde "t", optional gefolgt von "rue".
 |           #  oder ...
  y(?:es)?   #   Finde "y", optional gefolgt von "es".
 |           #  oder ...
  ja?        #   Finde "j", optional gefolgt von "a".
 |           #  oder ...
  ok(?:ay)?  #   Finde "ok", optional gefolgt von "ay".
)            # Ende der nicht-einfangenden Gruppe.
$            # Position am Ende des Strings sicherstellen.
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert, Freiform
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Diese Regex ist im Prinzip nur ein einfacher Test auf einen von neun literalen Werten, wobei die Groß-/Kleinschreibung ignoriert wird. Sie könnte auch anders geschrieben werden. So ist zum Beispiel ‹`^(?:[1tyj]|true|yes|ja|ok(?:ay)?)$`› ein ebenso guter Ansatz. Man könnte auch einfach eine Alternation mit allen neun Werten nutzen, wie zum Beispiel ‹`^(?:1|t|true|y|yes|j|ja|ok|okay)$`›, aber aus Performancegründen ist es im Allgemeinen besser, die Anzahl der Alternativen mit dem Pipe-Operator ‹`|`› zu reduzieren und Zeichenklassen und optionale Endungen (mit dem Quantor ‹`?`›) vorzuziehen. In diesem Fall

ist der Performanceunterschied vermutlich nur minimal, aber es ist nicht verkehrt, die Performance von Regexes immer im Hinterkopf zu behalten. Manchmal kann der Unterschied zwischen den verschiedenen Vorgehensweisen erstaunlich groß sein.

Alle diese Beispiele umgeben die möglichen Werte mit einer nicht-einfangenden Gruppe, um die Reichweite des Alternationsoperators zu begrenzen. Würden wir die Gruppe weglassen und stattdessen so etwas wie ‹^true|yes$› verwenden, würde die Regex-Engine nach „dem Anfang des Strings, gefolgt von 'true', oder 'yes', gefolgt vom Ende des Strings" suchen. ‹^(?:true|yes)$› weist die Regex-Engine an, den Anfang des Strings zu finden, dann entweder „true" oder „yes" und dann das Ende des Strings.

Siehe auch

Rezepte 5.2 und 5.3.

4.12 US-Sozialversicherungsnummern validieren

Problem

Sie müssen prüfen, ob jemand eine gültige US-Sozialversicherungsnummer eingegeben hat.

Lösung

Wenn Sie nur sicherstellen wollen, dass sich ein String an das grundlegende Sozialversicherungsnummerformat hält und keine offensichtlich ungültigen Zahlen enthalten sind, stellt die folgende Regex eine einfache Lösung bereit. Brauchen Sie eine strengere Prüfung, die auch bei der Social Security Administration prüft, ob die Nummer zu einer lebenden Person gehört, werfen Sie einen Blick auf die Links im Abschnitt „Siehe auch" dieses Rezepts.

Regulärer Ausdruck

```
^(?!000|666)(?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))-: "
(?!00)[0-9]{2}-(?!0000)[0-9]{4}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Python

```
if re.match(r"^(?!000|666)(?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))-: "
(?!00)[0-9]{2}-(?!0000)[0-9]{4}$", sys.argv[1]):
    print "SSN ist gültig"
else:
    print "SSN ist ungültig"
```

Andere Programmiersprachen

In Rezept 3.5 finden Sie Informationen über das Implementieren dieses regulären Ausdrucks in anderen Programmiersprachen.

Diskussion

Sozialversicherungsnummern in den USA sind neunstellige Nummern im Format *AAA-GG-SSSS*:

> Die ersten drei Ziffern werden anhand der geografischen Region zugewiesen. Dies ist die *Area Number*. Die Area Number kann nicht den Wert 000 oder 666 haben, zudem gibt es aktuell keine Sozialversicherungsnummer mit einer Area Number größer als 772.
>
> Die Ziffern vier und fünf bilden die *Group Number* und liegen im Bereich 01 bis 99.
>
> Die letzten vier Ziffern sind *Serial Numbers* von 0001 bis 9999.

Dieses Rezept nutzt alle diese Regeln. Hier noch einmal der reguläre Ausdruck, dieses Mal Stück für Stück erläutert:

```
^            # Position am Anfang des Strings sicherstellen.
(?!000|666)  # Weder "000" noch "666" dürfen hier vorkommen.
(?:          # Gruppieren, aber nicht einfangen ...
  [0-6]      #   Ein Zeichen im Bereich von "0" bis "6" finden.
  [0-9]{2}   #   Zwei Ziffern finden.
 |           #  oder ...
  7          #   Eine literale "7" finden.
  (?:        #   Gruppieren, aber nicht einfangen ...
    [0-6]    #     Eine Ziffer im Bereich von "0" bis "6" finden.
    [0-9]    #     Eine Ziffer finden.
   |         #    oder ...
    7        #     Eine literale "7" finden.
    [0-2]    #     Eine Ziffer im Bereich von "0" bis "2" finden.
  )          #   Ende der nicht-einfangenden Gruppe.
)            # Ende der nicht-einfangenden Gruppe.
-            # Einen literalen "-" finden.
(?!00)       # Sicherstellen, dass "00" hier nicht vorkommt.
[0-9]{2}     # Zwei Ziffern finden.
-            # Einen literalen "-" finden.
(?!0000)     # Sicherstellen, dass "0000" hier nicht vorkommt.
[0-9]{4}     # Vier Ziffern finden.
$            # Position am Ende des Strings sicherstellen.
```

Regex-Optionen: Freiform
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Abgesehen von den Tokens ‹^› und ‹$›, die sicherstellen, dass die Position am Anfang und Ende des Texts gefunden wird, kann diese Regex in drei Zifferngruppen aufgeteilt werden, die durch Bindestriche getrennt sind. Die erste Gruppe ist die komplexeste. Die zweite und die dritte Gruppe passen einfach auf zwei beziehungsweise vier Ziffern. Dabei

wird aber vorher ein negatives Lookahead genutzt, um zu verhindern, dass alle Ziffern solch einer Gruppe 0 sind.

Die erste Zifferngruppe ist viel komplexer und auch schlechter lesbar, denn sie passt auf einen ganzen Bereich. Zunächst wird das negative Lookahead ‹(?!000|666)› verwendet, um die Werte „000“ und „666“ auszuschließen. Als Nächstes geht es darum, alle Zahlen größer als 772 auszuschließen.

Da reguläre Ausdrücke mit Text arbeiten und nicht mit Zahlen, müssen wir den Bereich Zeichen für Zeichen unterteilen. Zunächst einmal wissen wir, dass jede dreistellige Zahl gültig ist, deren erste Ziffern im Bereich von 0 bis 6 liegt. Durch das vorherige negative Lookahead sind die ungültigen Zahlen 000 und 666 schon ausgeschlossen. Dieser erste Teil wird recht einfach durch ein paar Zeichenklassen und einen Quantor umgesetzt: ‹[0-6][0-9]{2}›. Da wir eine Alternative für Zahlen benötigen, die mit 7 beginnen, nutzen wir eine Gruppe wie in ‹(?:[0-6][0-9]{2}|7)›, um die Reichweite des Alternationsoperators einzuschränken.

Nummern, die mit 7 beginnen, sind nur zulässig, wenn sie im Bereich von 700 bis 772 liegen, daher müssen wir nun die Nummern in Abhängigkeit von der zweiten Ziffern unterteilen. Liegt sie zwischen 0 und 6, ist eine beliebige dritte Ziffer erlaubt. Ist die zweite Ziffer 7, muss die dritte Ziffer zwischen 0 und 2 liegen. Bringen wir alle diese Regeln zusammen, erhalten wir ‹7(?:[0-6][0-9]|7[0-2])›.

Fügen Sie das schließlich in die äußere Gruppe für die restlichen gültigen Nummern ein, erhalten Sie ‹(?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))›. Das ist alles. Sie haben erfolgreich eine Regex erstellt, die eine dreistellige Nummer zwischen 000 und 772 findet.

Variationen

Sozialversicherungsnummern in Dokumenten finden

Wenn Sie in einem größeren Dokument nach Sozialversicherungsnummern suchen, ersetzen Sie die Anker ‹^› und ‹$› durch Wortgrenzen. Regex-Engines betrachten alle alphanumerischen Zeichen und den Unterstrich als Wortzeichen.

```
\b(?!000|666)(?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))-: "
(?!00)[0-9]{2}-(?!0000)[0-9]{4}\b
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Siehe auch

Die Website der Social Security Administration (*http://www.socialsecurity.gov*) beantwortet die am häufigsten gestellten Fragen und führt auch aktuelle Listen mit bisher zugewiesenen Area und Group Numbers.

Der Social Security Number Verification Service (SSNVS) unter *http://www.socialsecurity.gov/employer/ssnv.htm* stellt zwei Wege bereit, um zu überprüfen, ob Namen und Sozialversicherungsnummern den Daten der Social Security Administration entsprechen.

Der Umgang mit Zahlenbereichen, einschließlich Beispielen für das Finden solcher Bereiche, wird in Rezept 6.5 detaillierter erläutert.

4.13 ISBN validieren

Problem

Sie müssen die Gültigkeit einer International Standard Book Number (ISBN) prüfen. Diese kann entweder im älteren ISBN-10- oder im aktuellen ISBN-13-Format vorliegen. Am Anfang soll optional eine ISBN-Kennung stehen, zudem können die Teile der ISBN optional durch Bindestriche oder Leerzeichen getrennt sein. `ISBN 978-0-596-52068-7`, `ISBN-13: 978-0-596-52068-7`, `978 0 596 52068 7`, `9780596520687`, `ISBN-10 0-596-52068-9` und `0-596-52068-9` sind allesamt Beispiele für gültige Eingaben.

Lösung

Sie können eine ISBN nicht allein mit einer Regex validieren, da die letzte Ziffer mit einem Prüfsummenalgorithmus berechnet wird. Die regulären Ausdrücke in diesem Abschnitt überprüfen das Format einer ISBN, während die folgenden Codebeispiele auch prüfen, ob die letzte Ziffer korrekt ist.

Reguläre Ausdrücke

ISBN-10:

```
^(?:ISBN(?:-10)?:?")?(?=[-0-9X"]{13}$|[0-9X]{10}$)[0-9]{1,5}[-"]?:"
(?:[0-9]+[-"]?){2}[0-9X]$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

ISBN-13:

```
^(?:ISBN(?:-13)?:?")?(?=[-0-9"]{17}$|[0-9]{13}$)97[89][-"]?[0-9]{1,5}:"
[-"]?(?:[0-9]+[-"]?){2}[0-9]$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

ISBN-10 oder ISBN-13:

```
^(?:ISBN(?:-1[03])?:?")?(?=[-0-9"]{17}$|[-0-9X"]{13}$|[0-9X]{10}$):"
(?:97[89][-"]?)?[0-9]{1,5}[-"]?(?:[0-9]+[-"]?){2}[0-9X]$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

JavaScript

```
// `regex` prüft auf die Formate ISBN-10 oder ISBN-13
var regex = /^(?:ISBN(?:-1[03])?:? )?(?=[-0-9 ]{17}$|[-0-9X ]{13}$|:"
[0-9X]{10}$)(?:97[89][- ]?)?[0-9]{1,5}[- ]?(?:[0-9]+[- ]?){2}[0-9X]$/;

if (regex.test(subject)) {
  // Nicht zugehörige ISBN-Elemente entfernen, dann in ein Array aufteilen
  var chars = subject.replace(/[^0-9X]/g, "").split("");
  // Letzte ISBN-Ziffer aus `chars` entfernen und `last` zuweisen
  var last  = chars.pop();
  var sum   = 0;
  var digit = 10;
  var check;

  if (chars.length == 9) {
    // ISBN-10-Prüfziffer berechnen
    for (var i = 0; i < chars.length; i++) {
      sum += digit * parseInt(chars[i], 10);
      digit -= 1;
    }
    check = 11 - (sum % 11);
    if (check == 10) {
      check = "X";
    } else if (check == 11) {
      check = "0";
    }
  } else {
    // ISBN-13-Prüfziffer berechnen
    for (var i = 0; i < chars.length; i++) {
      sum += (i % 2 * 2 + 1) * parseInt(chars[i], 10);
    }
    check = 10 - (sum % 10);
    if (check == 10) {
      check = "0";
    }
  }

  if (check == last) {
    alert("Gültige ISBN");
  } else {
    alert("Ungültige ISBN-Prüfziffer");
  }
} else {
  alert("Ungültige ISBN");
}
```

Python

```
import re
import sys

# `regex` prüft auf die Formate ISBN-10 oder ISBN-13
regex = re.compile("^(?:ISBN(?:-1[03])?:? )?(?=[-0-9 ]{17}$|:"
[-0-9X ]{13}$|[0-9X]{10}$)(?:97[89][- ]?)?[0-9]{1,5}[- ]?:"
```

```
(?:[0-9]+[- ]?){2}[0-9X]$")

subject = sys.argv[1]

if regex.search(subject):
  # Nicht zugehörige ISBN-Elemente entfernen, dann in ein Array aufteilen
  chars = re.sub("[^0-9X]", "", subject).split("")
  # Letzte ISBN-Ziffer aus `chars` entfernen und `last` zuweisen
  last  = chars.pop()

  if len(chars) == 9:
    # ISBN-10-Prüfziffer berechnen
    val = sum((x + 2) * int(y) for x,y in enumerate(reversed(chars)))
    check = 11 - (val % 11)
    if check == 10:
      check = "X"
    elif check == 11:
      check = "0"
  else:
    # ISBN-13-Prüfziffer berechnen
    val = sum((x % 2 * 2 + 1) * int(y) for x,y in enumerate(chars))
    check = 10 - (val % 10)
    if check == 10:
      check = "0"

  if (str(check) == last):
    print "Gültige ISBN"
  else:
    print "Ungültige ISBN-Prüfziffer"
else:
  print "Ungültige ISBN"
```

Andere Programmiersprachen

In Rezept 3.5 wird beschrieben, wie Sie diesen regulären Ausdruck in anderen Programmiersprachen implementieren können.

Diskussion

Eine ISBN ist eine eindeutige Kennung für Bücher und bücherähnliche Produkte. Das zehnstellige ISBN-Format wurde als internationaler Standard ISO 2108 im Jahr 1970 veröffentlicht. Alle ISBN, die seit dem 1. Januar 2007 zugewiesen werden, sind 13-stellig.

ISBN-10- und ISBN-13-Nummern werden in vier beziehungsweise fünf Elemente aufgeteilt. Drei der Elemente haben eine variable, die verbleibenden ein oder zwei Elemente haben eine feste Länge. Alle fünf Teile werden normalerweise durch Bindestriche oder Leerzeichen getrennt. Dabei haben die Elemente folgende Bedeutung:

- 13-stellige ISBN beginnen mit dem Präfix 978 oder 979.
- Die *Gruppennummer* steht für einen geografisch oder sprachlich zusammenhängenden Raum. Sie kann eine bis fünf Ziffern lang sein.

- Die *Verlagsnummer* kann unterschiedlich lang sein und wird von der nationalen ISBN-Agentur vergeben.
- Die *Titelnummer* kann auch unterschiedlich lang sein und wird vom Verlag festgelegt.
- Das letzte Zeichen ist die *Prüfziffer*. Sie wird mit einem Prüfsummenalgorithmus ermittelt. Eine ISBN-10-Prüfziffer kann entweder die Werte 0 bis 9 oder den Buchstaben X (für die römische 10) enthalten. Eine ISBN-13-Prüfziffer liegt im Bereich von 0 bis 9. Die hier verwendeten Zeichen sind unterschiedlich, weil auch unterschiedliche Prüfsummenalgorithmen genutzt werden.

Die Regex für ISBN-10- und ISBN-13-Nummern wird im folgenden Beispiel in ihre Bestandteile zerlegt. Da sie hier im Freiform-Modus genutzt wird, wurden die literalen Leerzeichen in der Regex durch Backslashs maskiert. Bei Java müssen im Freiform-Modus Leerzeichen selbst in Zeichenklassen maskiert werden:

```
^                # Position am Anfang des Strings sicherstellen.
(?:              # Gruppieren, aber nicht einfangen ...
  ISBN           #   Den Text "ISBN" finden.
  (?:-1[03])?    #   Optional den Text "-10" oder "-13" finden.
  :?             #   Optional ein literales ":" finden.
  \              #   Ein (maskiertes) Leerzeichen finden.
)?               # Die Gruppe null oder ein Mal finden.
(?=              # Sicherstellen, dass das Folgende passt ...
  [-0-9\ ]{17}$  #   17 Bindestriche, Ziffern und Leerzeichen bis zum Ende
 |               #     des Strings finden. Oder ...
  [-0-9X\ ]{13}$ #   13 Bindestriche, Ziffern, X und Leerzeichen bis zum Ende
 |               #     des Strings finden. Oder ...
  [0-9X]{10}$    #   10 Ziffern und X bis zum Ende finden.
)                # Ende des positiven Lookahead.
(?:              # Gruppieren, aber nicht einfangen ...
  97[89]         #   Den Text "978" oder "979" finden.
  [-\ ]?         #   Optional einen Bindestrich oder ein Leerzeichen finden.
)?               # Die Gruppe null oder ein Mal finden.
[0-9]{1,5}       # Eine Ziffer ein bis vier Mal finden.
[-\ ]?           # Optional einen Bindestrich oder ein Leerzeichen finden.
(?:              # Gruppieren, aber nicht einfangen ...
  [0-9]+         #   Eine Ziffer ein Mal oder häufiger finden.
  [-\ ]?         #   Optional einen Bindestrich oder ein Leerzeichen finden.
){2}             # Die Gruppe genau zwei Mal finden.
[0-9X]           # Eine Ziffer oder ein "X" finden.
$                # Position am Ende des Strings sicherstellen.
```

Regex-Optionen: Freiform
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Der erste Teil ‹(?:ISBN(?:-1[03])?:?”)?› hat drei optionale Elemente, durch die einer der folgenden sieben Strings passt (mit Ausnahme des leeren Strings enthalten alle ein Leerzeichen am Ende):

- ISBN”
- ISBN-10”
- ISBN-13”

- ISBN:”
- ISBN-10:”
- ISBN-13:”
- *Ein leerer String (kein Präfix)*

Als Nächstes sorgt das positive Lookahead ‹(?=[-0-9”]{17}$|[-0-9X”]{13}$|[0-9X]{10}$)› dafür, dass eine der drei Optionen (getrennt durch den Alternationsoperator ‹|›) bezüglich der Länge und erlaubten Zeichen für den Rest der Übereinstimmung gültig ist. Alle drei Optionen enden mit dem Anker ‹$›. Dadurch ist sichergestellt, dass es keinen nachfolgenden Text gibt, der nicht zu einem der Muster passt:

‹[-0-9”]{17}$›
: Erlaubt eine ISBN-13 mit vier Trennzeichen (insgesamt 17 Zeichen).

‹[-0-9X”]{13}$›
: Erlaubt eine ISBN-13 ohne Trennzeichen oder eine ISBN-10 mit drei Trennzeichen (insgesamt 13 Zeichen).

‹[0-9X]{10}$›
: Erlaubt eine ISBN-10 ohne Trennzeichen (insgesamt 10 Zeichen).

Nachdem das positive Lookahead die Länge und die Zeichen geprüft hat, können wir die einzelnen Elemente der ISBN auswerten, ohne uns über ihre Gesamtlänge Gedanken machen zu müssen. ‹(?:97[89][-”]?)?› passt zum Präfix „978“ oder „979“, das von einer ISBN-13 gefordert wird. Die nicht-einfangende Gruppe ist optional, weil sie bei einer ISBN-10 nicht vorkommt. ‹[0-9]{1,5}[-”]?› passt zu einer Zifferngruppe mit ein bis fünf Ziffern, denen optional ein Trennzeichen folgt. ‹(?:[0-9]+[-”]?){2}› passt zur Verlags- und Titelnummer und deren optionalen Separatoren. Schließlich passt ‹[0-9X]$› auf die Prüfziffer am Ende des Strings.

Ein regulärer Ausdruck kann zwar prüfen, ob die letzte Ziffer ein gültiges Zeichen nutzt (eine Ziffer oder ein X), aber nicht, ob es sich dabei um die korrekte Prüfziffer handelt. Einer der beiden Prüfsummenalgorithmen (abhängig davon, ob Sie mit einer ISBN-10- oder einer ISBN-13-Nummer arbeiten) wird verwendet, um wenigstens halbwegs sicher zu sein, dass die ISBN-Ziffern nicht unabsichtlich vertauscht oder anders falsch eingegeben wurden. Der weiter oben gezeigte Beispielcode für JavaScript und Python implementiert beide Algorithmen. Der folgende Abschnitt beschreibt die Prüfsummenregeln, damit Sie diese Algorithmen auch in anderen Programmiersprachen implementieren können.

ISBN-10-Prüfsumme

Die Prüfziffer einer ISBN-10-Nummer kann den Wert 0 bis 10 haben (wobei die römische Zahl X statt der 10 verwendet wird). Sie wird wie folgt ermittelt:

1. Multipliziere jede der ersten 9 Ziffern mit einer Zahl in der absteigenden Folge von 10 bis 2 und addiere die Ergebnisse.
2. Teile die Summe durch 11.

3. Ziehe den Rest (nicht den Quotienten) von 11 ab.
4. Wenn das Ergebnis 11 ist, verwende die Ziffer 0; ist es 10, verwende den Buchstaben X.

Hier ein Beispiel, wie man die ISBN-10-Prüfziffer für `3-89721-957-?` ermittelt:

```
Schritt 1:
sum = 10×3 + 9×8 + 8×9 + 7×7 + 6×2 + 5×1 + 4×9 + 3×5 + 2×7
    =   30 +  72 +  72 +  49 +  18 +   5 +  36 +  15 +  14
    = 305
Schritt 2:
    305 ÷ 11 = 27, Rest 8
Schritt 3:
    11 - 8 = 3
Schritt 4:
    3 [keine Ersetzung notwendig]
```

Die Prüfziffer ist 3, daher ist die komplette ISBN `ISBN 3-89721-957-3`.

ISBN-13-Prüfsumme

Eine ISBN-13-Prüfziffer liegt im Bereich von 0 bis 9 und wird in ähnlichen Schritten ermittelt.

Multipliziere jede der ersten 12 Ziffern mit 1 oder 3 – immer abwechselnd von links nach rechts – und addiere die Ergebnisse.

Teile die Summe durch 10.

Ziehe den Rest (nicht den Quotienten) von 10 ab.

Ist das Ergebnis 10, verwende die Ziffer 0.

So wird zum Beispiel die ISBN-13-Prüfziffer für `978-3-89721-957-?` wie folgt berechnet:

```
Schritt 1:
sum = 1×9 + 3×7 + 1×8 + 3×3 + 1×8 + 3×9 + 1×7 + 3×2 + 1×1 + 3×9 + 1×5 + 3×7
    =   9 +  21 +   8 +   9 +   8 +  27 +   7 +   6 +   1 +  27 +   5 +  21
    = 149
Schritt 2:
    149 ÷ 10 = 14, remainder 9
Schritt 3:
    10 - 9 = 1
Schritt 4:
    1 [keine Ersetzung notwendig]
```

Die Prüfziffer ist 1, und die komplette ISBN hat den Wert `ISBN 978-3-89721-957-1`.

Variationen

ISBNs in Dokumenten finden

Diese Version der Regex für ISBN-10 und ISBN-13 nutzt statt der Anker Wortgrenzen, um ISBN in längeren Texten zu finden, dabei aber sicherzustellen, dass sie für sich ste-

hen. Der Text „ISBN“ ist in dieser Regex immer erforderlich. Das hat zwei Gründe. Zum einen verhindert man so fälschlicherweise als ISBN erkannte Zahlenfolgen (denn die Regex könnte potenziell beliebige 10- oder 13-stellige Zahlen finden), und zum anderen sollen ISBN diesen Text offiziell enthalten, wenn sie ausgegeben werden:

```
\bISBN(?:-1[03])?:?”(?=[-0-9”]{17}$|[-0-9X”]{13}$|[0-9X]{10}$): "
(?:97[89][-”]?)?[0-9]{1,5}[-”]?(?:[0-9]+[-”]?){2}[0-9X]\b
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Falsche ISBN-Kennungen finden

Die vorigen Regexes haben ein Problem: Man kann den Text „ISBN-13“ haben, auf den dann aber eine ISBN-10-Nummer folgt und umgekehrt. Die folgende Regex nutzt Regex-Bedingungen (siehe Rezept 2.17), um sicherzustellen, dass eine Kennung „ISBN-10“ oder „ISBN-13“ immer vom passenden ISBN-Typ begleitet wird. Wenn der Typ nicht explizit angegeben ist, sind beide Nummernarten möglich. Diese Regex ist in den meisten Fällen doch etwas übertrieben, da man das gleiche Ergebnis auch einfacher erreichen kann, wenn man die getrennten ISBN-10- und ISBN-13-Regexes nutzt. Sie soll hier eher gezeigt werden, um eine interessante Anwendung von regulären Ausdrücken zu demonstrieren:

```
^
(?:ISBN(-1(?:(0)|3))?:?\ )?
(?(1)
  (?(2)
    (?=[-0-9X ]{13}$|[0-9X]{10}$)
    [0-9]{1,5}[- ]?(?:[0-9]+[- ]?){2}[0-9X]$
   |
    (?=[-0-9 ]{17}$|[0-9]{13}$)
    97[89][- ]?[0-9]{1,5}[- ]?(?:[0-9]+[- ]?){2}[0-9]$
  )
 |
  (?=[-0-9 ]{17}$|[-0-9X ]{13}$|[0-9X]{10}$)
  (?:97[89][- ]?)?[0-9]{1,5}[- ]?(?:[0-9]+[- ]?){2}[0-9X]$
)
$
```

Regex-Optionen: Freiform
Regex-Varianten: .NET, PCRE, Perl, Python

Siehe auch

Die aktuellste Version der ISBN-Dokumente lassen sich auf der Website der International ISBN Agency unter *http://www.isbn-international.org* finden.

Die offizielle Liste mit Gruppennummern findet sich ebenfalls auf der Website der International ISBN Agency. Anhand dieser Liste können Sie das Ursprungsland eines Buchs mithilfe der ersten 1 bis 5 Ziffern der ISBN ermitteln.

4.14 ZIP-Codes validieren

Problem

Sie müssen einen ZIP-Code (eine US-Postleitzahl) validieren, wobei sowohl das fünfstellige als auch das neunstellige Format (*ZIP + 4*) zu erkennen ist. Die Regex sollte auf `12345` und `12345-6789` passen, aber nicht auf `1234`, `123456`, `123456789` oder `1234-56789`.

Lösung

Regulärer Ausdruck

```
^[0-9]{5}(?:-[0-9]{4})?$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

VB.NET

```
If Regex.IsMatch(subjectString, "^[0-9]{5}(?:-[0-9]{4})?$") Then
    Console.WriteLine("Gültiger ZIP-Code")
Else
    Console.WriteLine("Ungültiger ZIP-Code")
End If
```

Andere Programmiersprachen

In Rezept 3.5 finden Sie Informationen über das Implementieren dieses regulären Ausdrucks in anderen Programmiersprachen.

Diskussion

Der reguläre Ausdruck für den ZIP-Code sieht im Freiform-Modus so aus:

```
^           # Position am Anfang des Strings sicherstellen.
[0-9]{5}    # Fünf Ziffern finden.
(?:         # Gruppieren, aber nicht einfangen ...
  -         #   Einen literalen "-" finden.
  [0-9]{4}  #   Vier Ziffern finden.
)           # Ende der nicht-einfangenden Gruppe.
  ?         #   Die vorige Gruppe null oder ein Mal finden.
$           # Position am Ende des Strings sicherstellen.
```

Regex-Optionen: Freiform
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Diese Regex ist ziemlich einfach, daher ist nicht sehr viel dazu zu sagen. Eine simple Änderung ermöglicht es Ihnen, ZIP-Codes in einem längeren String zu finden: Ersetzen Sie die Anker ‹`^`› und ‹`$`› durch Wortgrenzen: ‹`\b[0-9]{5}(?:-[0-9]{4})?\b`›.

Siehe auch

Rezepte 4.15, 4.16 und 4.17.

4.15 Kanadische Postleitzahlen validieren

Problem

Sie wollen prüfen, ob ein String eine kanadische Postleitzahl enthält.

Lösung

```
^(?!.*[DFIOQU])[A-VXY][0-9][A-Z]”[0-9][A-Z][0-9]$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Diskussion

Das negative Lookahead am Anfang dieses regulären Ausdrucks verhindert, dass sich irgendwo im Ausgangstext die Buchstaben D, F, I, O, Q oder U befinden. Die Zeichenklasse `‹[A-VXY]›` verhindert darüber hinaus, dass W oder Z das erste Zeichen ist. Neben diesen beiden Ausnahmen werden für kanadische Postleitzahlen einfach abwechselnde Folgen von sechs alphanumerischen Zeichen genutzt, wobei in der Mitte ein Leerzeichen steht. So passt diese Regex zum Beispiel auf `K1A 0B1`. Dabei handelt es sich um die Postleitzahl für die Zentrale der kanadischen Post in Ottawa.

Siehe auch

Rezepte 4.14, 4.16 und 4.17.

4.16 Britische Postleitzahlen validieren

Problem

Sie benötigen einen regulären Ausdruck, der britische Postleitzahlen erkennt.

Lösung

```
^[A-Z]{1,2}[0-9R][0-9A-Z]?”[0-9][ABD-HJLNP-UW-Z]{2}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Diskussion

Postleitzahlen in Großbritannien (oder auch *Postcodes*, wie sie dort genannt werden) bestehen aus fünf bis sieben alphanumerischen Zeichen, die durch ein Leerzeichen unterteilt sind. Die Regeln legen fest, welche Zeichen an welcher Position stehen dürfen. Leider sind sie ziemlich kompliziert und voller Ausnahmen. Daher kümmert sich dieser reguläre Ausdruck nur um die grundlegenden Regeln.

Siehe auch

British Standard BS7666, verfügbar unter *http://www.govtalk.gov.uk/gdsc/html/frames/PostCode.htm*. Hier werden die Regeln für britische Postleitzahlen beschrieben.

Rezepte 4.14, 4.15 und 4.17.

4.17 Deutsche Postleitzahlen validieren

Problem

Sie benötigen einen regulären Ausdruck, der deutsche Postleitzahlen erkennt.

Lösung

```
^[0-9]{5}$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Diskussion

Deutsche Postleitzahlen bestehen einfach aus fünf Ziffern ohne weitere Unterteilung. Beachten Sie, dass Postleitzahlen bei einer weiteren Verarbeitung nicht als Zahlen angesehen werden sollten, sondern eher als Zeichenkette. In Deutschland gibt es eine Reihe von Orten, deren Postleitzahl mit einer 0 beginnt. Speichert man Postleitzahlen als Zahl, verschwindet diese 0, was verwirrt und eventuell dafür sorgt, dass die Post nicht (direkt) ankommt.

Variationen

Postleitzahlen in anderen europäischen Ländern

Belgien

```
‹^[1-9][0-9]{3}$›
```

Bulgarien

```
‹^[1-9][0-9]{3}$›
```

Dänemark

```
‹^[1-9][0-9]{3}$›
```

Finnland
    ‹^[0-9]{5}$›
Frankreich und Monaco
    ‹^[0-9]{5}$›
Griechenland
    ‹^[1-8][0-9]{4}$›
Italien, San Marino und Vatikanstadt
    ‹^[0-9]{5}$›
Kroatien
    ‹^(?:[1-4][0-9]|5[1-3])[0-9]{3}$›
Montenegro
    ‹^8[145][0-9]{3}$›
Niederlande
    ‹^[1-9][0-9]{3}"?[A-Z]{2}$›
Norwegen
    ‹^[0-9]{4}$›
Österreich
    ‹^[1-9][0-9]{3}$›
Polen
    ‹^[0-9]{2}-[0-9]{3}$›
Portugal
    ‹^[1-9][0-9]{3}-?[0-9]{2}$›
Rumänien
    ‹^[0-9]{6}$›
Schweden
    ‹^[1-9][0-9]{2}"?[0-9]{2}$›
Schweiz und Liechtenstein
    ‹^[1-9][0-9]{3}$›
Slowakei
    ‹^[890][0-9]{2}"?[0-9]{2}$›
Spanien
    ‹^(?:0[1-9]|[1-4][0-9]|5[12])[0-9]{3}$›
Tschechien
    ‹^[1-7][0-9]{2}"?[0-9]{2}$›
Ungarn
    ‹^[1-9][0-9]{3}$›
Zypern
    ‹^[1-9][0-9]{3}$›

Siehe auch

Rezepte 4.14, 4.15 und 4.16.

4.18 Namen von „Vorname Nachname“ nach „Nachname, Vorname“ umwandeln

Problem

Sie wollen Personennamen von „Vorname Nachname“ umwandeln in „Nachname, Vorname“, um so alphabetisch sortieren zu können. Zudem wollen Sie zusätzlich auf andere Namensbestandteile Rücksicht nehmen, zum Beispiel auf einen zweiten Vornamen.

Lösung

Leider ist es nicht möglich, Namen mit einem regulären Ausdruck zuverlässig zu parsen. Reguläre Ausdrücke sind strikt, während Namen so flexibel gehandhabt werden, dass selbst Menschen durcheinanderkommen. Das Bestimmen der Struktur eines Namens und die richtige Einordnung in eine alphabetisch sortierte Liste erfordern häufig das Einbeziehen traditioneller und landesspezifischer Konventionen, und selbst persönliche Vorlieben können eine Rolle spielen. Wenn Sie aber dazu bereit sind, gewissen Annahmen über Ihre Daten zu treffen und auch dann und wann Fehler akzeptieren können, kann ein regulärer Ausdruck eine schnelle Lösung bieten.

Der folgende reguläre Ausdruck wird eher einfach gehalten und soll nicht unbedingt alle möglichen Grenzfälle abdecken.

Regulärer Ausdruck

```
^(.+?)”([^\s]+)$
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ersetzung

```
$2,”$1
```

Ersetzungstextvarianten: .NET, Java, JavaScript, Perl, PHP

```
\2,”\1
```

Ersetzungstextvarianten: Python, Ruby

JavaScript

```
function formatName (name) {
    return name.replace(/^(.+?)”([^\s,]+)$/i,
                        "$2, $1");
}
```

Andere Programmiersprachen

In Rezept 3.15 finden Sie Informationen über das Implementieren dieses regulären Ausdrucks in anderen Programmiersprachen.

Diskussion

Lassen Sie uns diesen regulären Ausdruck erst mal Stück für Stück betrachten. Danach erklären wir Ihnen, welche Teile eines Namens von welchen Regex-Elementen gefunden werden. Da die Regex hier im Freiform-Modus geschrieben ist, wurden die literalen Leerzeichen durch Backslashs maskiert:

```
^             # Position am Anfang des Strings sicherstellen.
(             # Gruppe für Rückwärtsreferenz 1 ...
  .+?         #   Eines oder mehrere Zeichen finden, aber so wenig wie möglich.
)             # Ende der einfangenden Gruppe.
\             # Ein Leerzeichen finden.
(             # Gruppe für Rückwärtsreferenz 2 ...
  [^\s]+      #   Eines oder mehrere Zeichen finden, die keine Leerzeichen sind.
)             # Ende der einfangenden Gruppe.
$             # Position am Ende des Strings sicherstellen.
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert, Freiform
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Dieser reguläre Ausdruck geht von folgenden Annahmen aus:

- Der Ausgangstext enthält mindestens einen Vornamen und einen Nachnamen (weitere Bestandteile sind optional).
- Der Vorname steht vor dem Nachnamen.

Ein paar Probleme gibt es aber:

- Der reguläre Ausdruck kann keine mehrteiligen Nachnamen erkennen, die nicht per Bindestrich verbunden sind. So würde `Sacha Baron Cohen` zum Beispiel durch `Cohen, Sacha Baron` ersetzt werden und nicht durch die korrekte Version `Baron Cohen, Sacha`.
- Namensbestandteile vor dem Familiennamen werden auch nicht dem Nachnamen zugeordnet, obwohl dies aufgrund von Konventionen oder persönlichen Vorlieben teilweise gewünscht wird (so kann „Charles de Gaulle“ entweder als „de Gaulle, Charles“ oder als „Gaulle, Charles de“ aufgeführt sein).
- Aufgrund der Anker `‹^›` und `‹$›`, die die Übereinstimmung mit dem Anfang und Ende des Strings verbinden, kann keine Ersetzung vorgenommen werden, wenn nicht der gesamte Ausgangstext zum Muster passt. Wird keine passende Übereinstimmung gefunden (weil zum Beispiel der Ausgangstext nur einen Namen enthält), bleibt der Name so bestehen.

Der reguläre Ausdruck nutzt zwei einfangende Gruppen, um den Namen aufzuteilen. Diese Elemente werden dann über Rückwärtsreferenzen in der gewünschten Reihenfolge

wieder zusammengesetzt. Die erste einfangende Gruppe nutzt das ausgesprochen flexible Muster ‹.+?›, um den ersten Vornamen zusammen mit allen weiteren Vornamen und den zusätzlichen Bestandteilen des Nachnamens einzufangen, wie zum Beispiel das deutsche „von“ oder das französische, portugiesische und spanische „de“. Diese Namenselemente werden zusammen bearbeitet, da sie in der Ausgabe auch nacheinander erscheinen sollen.

Die zweite einfangende Gruppe passt durch ‹[^\s]+› auf den Nachnamen. Wie beim Punkt in der ersten einfangenden Gruppe ermöglicht die Flexibilität dieser Zeichenklasse auch die Verwendung von Umlauten und anderen exotischen Zeichen.

In Tabelle 4-2 werden ein paar Beispiele für mit dieser Regex und dem entsprechenden Ersetzungstext umgestellte Namen aufgeführt.

Tabelle 4-2: Formatierte Namen

Eingabe	Ausgabe
Robert Downey	Downey, Robert
John F. Kennedy	Kennedy, John F.
Scarlett O'Hara	O'Hara, Scarlett
Pepé Le Pew	Pew, Pepé Le
J.R.R. Tolkien	Tolkien, J.R.R.
Catherine Zeta-Jones	Zeta-Jones, Catherine

Variationen

Nachnamensbestandteile am Anfang des Namens aufführen

Im folgenden regulären Ausdruck haben wir ein Element ergänzt, durch das zusätzliche Bestandteile des Nachnamens bei ihm verbleiben. Diese Regex berücksichtigt „de“, „du“, „la“, „le“, „St“, „St.“, „Ste“, „Ste.“, „van“ und „von“. Dabei sind auch mehrere solcher Bestandteile möglich (zum Beispiel „de la”):

```
^(.+?)”((?:(?:D[eu]|L[ae]|Ste?\.?|V[ao]n)”)*[^\s]+)$
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
$2,”$1
```

Ersetzungstextvarianten: .NET, Java, JavaScript, Perl, PHP

```
\2,”\1
```

Ersetzungstextvarianten: Python, Ruby

4.19 Kreditkartennummern validieren

Problem

Sie sollen für eine Firma ein Bestellformular bauen, das auch eine Bezahlung per Kreditkarte zulässt. Da die Karten-Servicegesellschaft für jeden Transaktionsversuch eine Gebühr erhebt – auch für fehlgeschlagene Versuche –, wollen Sie mit einem regulären Ausdruck die offensichtlich ungültigen Kreditkartennummern ausfiltern.

Nebenbei verbessert das auch die Bedienungsfreundlichkeit. Ein regulärer Ausdruck kann offensichtliche Tippfehler sofort erkennen, sobald der Anwender mit dem Ausfüllen der Felder auf der Webseite fertig ist. Eine Anfrage bei der Karten-Servicegesellschaft dauert dagegen leicht einmal 10 bis 30 Sekunden.

Lösung

Leerzeichen und Bindestriche entfernen

Lesen Sie die vom Kunden eingegebene Kreditkartennummer aus und speichern Sie sie in einer Variablen. Bevor Sie die Gültigkeit der Nummer überprüfen, suchen Sie nach Leerzeichen und Bindestrichen und entfernen sie aus der Nummer. Ersetzen Sie diesen regulären Ausdruck global durch einen leeren Ersetzungstext:

```
[ -]
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

In Rezept 3.14 ist beschrieben, wie Sie diese erste Ersetzung vornehmen.

Validieren der Nummer

Nachdem Leerzeichen und Bindestriche aus der Eingabe entfernt wurden, prüft dieser reguläre Ausdruck, ob die Kreditkartennummer dem Format einer der sechs großen Kreditkartenfirmen entspricht. Dabei nutzt die Regex benannte Captures, um herauszufinden, was für eine Kreditkarte der Kunde hat:

```
^(?:
(?<visa>4[0-9]{12}(?:[0-9]{3})?) |
(?<mastercard>5[1-5][0-9]{14}) |
(?<discover>6(?:011|5[0-9][0-9])[0-9]{12}) |
(?<amex>3[47][0-9]{13}) |
(?<diners>3(?:0[0-5]|[68][0-9])[0-9]{11}) |
(?<jcb>(?:2131|1800|35\d{3})\d{11})
)$
```

Regex-Optionen: Freiform
Regex-Varianten: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
^(?:
(?P<visa>4[0-9]{12}(?:[0-9]{3})?) |
(?P<mastercard>5[1-5][0-9]{14}) |
(?P<discover>6(?:011|5[0-9][0-9])[0-9]{12}) |
(?P<amex>3[47][0-9]{13}) |
(?P<diners>3(?:0[0-5]|[68][0-9])[0-9]{11}) |
(?P<jcb>(?:2131|1800|35\d{3})\d{11})
)$
```

Regex-Optionen: Freiform
Regex-Varianten: PCRE, Python

Java, Perl 5.6, Perl 5.8 und Ruby 1.8 unterstützen keine benannten Captures. Hier können Sie nummerierte Captures verwenden. Gruppe 1 fängt die Visa-Karten, Gruppe 2 die MasterCards und so weiter bis zur Gruppe 6 für JCB:

```
^(?:
(4[0-9]{12}(?:[0-9]{3})?) |         # Visa
(5[1-5][0-9]{14}) |                 # MasterCard
(6(?:011|5[0-9][0-9])[0-9]{12}) |   # Discover
(3[47][0-9]{13}) |                  # American Express
(3(?:0[0-5]|[68][0-9])[0-9]{11}) |  # Diners Club
((?:2131|1800|35\d{3})\d{11})       # JCB
)$
```

Regex-Optionen: Freiform
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

JavaScript unterstützt keinen Freiform-Modus. Entfernen wir den Leerraum und die Kommentare, erhalten wir:

```
^(?:(4[0-9]{12}(?:[0-9]{3})?)|(5[1-5][0-9]{14})|: "
(6(?:011|5[0-9][0-9])[0-9]{12})|(3[47][0-9]{13})|: "
(3(?:0[0-5]|[68][0-9])[0-9]{11})|((?:2131|1800|35\d{3})\d{11}))$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Wenn Sie nicht wissen müssen, um welchen Kartentyp es geht, können Sie die unnötigen einfangenden Gruppen entfernen:

```
^(?:
4[0-9]{12}(?:[0-9]{3})? |         # Visa
5[1-5][0-9]{14} |                 # MasterCard
6(?:011|5[0-9][0-9])[0-9]{12} |   # Discover
3[47][0-9]{13} |                  # American Express
3(?:0[0-5]|[68][0-9])[0-9]{11} |  # Diners Club
(?:2131|1800|35\d{3})\d{11}       # JCB
)$
```

Regex-Optionen: Freiform
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Für JavaScript:

```
^(?:4[0-9]{12}(?:[0-9]{3})?|5[1-5][0-9]{14}|6(?:011|5[0-9][0-9])[0-9]{12}|
3[47][0-9]{13}|3(?:0[0-5]|[68][0-9])[0-9]{11}|(?:2131|1800|35\d{3})\d{11})$
```

Regex-Optionen: Keine

Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Folgen Sie der Anleitung in Rezept 3.6, um Ihrem Bestellformular diesen regulären Ausdruck hinzuzufügen und die Kreditkartennummer zu überprüfen. Wenn Sie unterschiedliche Serviceunternehmen für verschiedene Karten nutzen oder einfach selbst eine Statistik führen wollen, können Sie wie in Rezept 3.9 prüfen, welche benannten oder nummerierten einfangenden Gruppen die Übereinstimmung enthalten. Damit erfahren Sie, von welcher Firma die Karte Ihres Kunden ist.

Beispiel-Webseite mit JavaScript

```
<html>
<head>
<title>Kreditkartentest</title>
</head>

<body>
<h1>Kreditkartentest</h1>

<form>
<p>Bitte geben Sie Ihre Kreditkartennummer ein:</p>

<p><input type="text" size="20" name="cardnumber"
  onkeyup="validatecardnumber(this.value)"></p>

<p id="notice">(keine Kartennummer eingegeben)</p>
</form>

<script>
function validatecardnumber(cardnumber) {
  // Leerzeichen und Bindestriche entfernen
  cardnumber = cardnumber.replace(/[ -]/g, '');
  // Prüfen, ob die Karte gültig ist
  // Die Regex fängt die Nummer in einer der einfangenden Gruppen
  var match = /^(?:(4[0-9]{12}(?:[0-9]{3})?)|(5[1-5][0-9]{14})|
(6(?:011|5[0-9][0-9])[0-9]{12})|(3[47][0-9]{13})|(3(?:0[0-5]|[68][0-9])
[0-9]{11})|((?:2131|1800|35\d{3})\d{11}))$/.exec(cardnumber);
  if (match) {
    // Liste der Kartentypen in der gleichen Reihenfolge wie die einfangenden Gruppen
    var types = ['Visa', 'MasterCard', 'Discover', 'American Express',
                 'Diners Club', 'JCB'];
    // Einfangende Gruppe finden, die passt
    // Das nullte Element des Match-Arrays überspringen (das Gesamtsuchergebnis)
    for (var i = 1; i < match.length; i++) {
      if (match[i]) {
        // Kartentyp für diese Gruppe anzeigen
        document.getElementById('notice').innerHTML = types[i - 1];
```

```
        break;
      }
    }
  } else {
    document.getElementById('notice').innerHTML = '(Ungültige Kartennummer)';
  }
}
</script>
</body>
</html>
```

Diskussion

Leerzeichen und Bindestriche entfernen

Auf Kreditkarten sind die in die Karte eingestanzten Ziffern meist in Vierergruppen unterteilt. So lässt sich die Kartennummer leichter lesen. Natürlich werden viele Leute versuchen, ihre Kartennummer auch genau so auf einer Webseite einzugeben – einschließlich der Leerzeichen.

Schreibt man einen regulären Ausdruck, der eine Kartennummer validieren soll und dabei Leerzeichen, Bindestriche und was auch immer berücksichtigen will, ist das deutlich schwieriger als einer, der nur Ziffern zulässt. Um daher den Kunden nicht damit zu nerven, dass er die Kartennummer nochmals ohne Leerzeichen oder Bindestriche eingeben soll, entfernen Sie sie einfach vor dem Überprüfen der Nummer und der Übermittlung an die Karten-Servicegesellschaft.

Der reguläre Ausdruck `‹["-]›` passt auf ein Leerzeichen oder einen Bindestrich. Ersetzen Sie alle Übereinstimmungen dieses regulären Ausdrucks durch einen leeren String, werden damit alle Leerzeichen und Bindestriche entfernt.

Kreditkartennummern können nur aus Ziffern bestehen. Statt mit `‹["-]›` lediglich Leerzeichen und Bindestriche zu entfernen, können Sie auch die Zeichenklassenabkürzung `‹\D›` nutzen, um alles zu entfernen, was keine Ziffer ist.

Validieren der Nummer

Jede Kreditkartenfirma verwendet ein anderes Nummernformat. Wir nutzen diese Unterschiede, damit der Anwender eine Nummer eingeben kann, ohne die Kartenfirma angeben zu müssen. Die Firma kann dann aus der Nummer ermittelt werden. Die Formate sind:

Visa
: 13 oder 16 Ziffern, beginnend mit einer `4`.

MasterCard
: 16 Ziffern, beginnend mit `51` bis `55`.

Discover
: 16 Ziffern, beginnend mit `6011` oder `65`.

American Express
: 15 Ziffern, beginnend mit 34 oder 37.

Diners Club
: 14 Ziffern, beginnend mit 300 bis 305, 36 oder 38.

JCB
: 15 Ziffern, beginnend mit 2131 oder 1800, oder 16 Ziffern, beginnend mit 35.

Wenn Sie nur bestimmte Kartenfirmen akzeptieren, können Sie die Karten aus der Regex entfernen, die Sie nicht haben wollen. Beim Entfernen von JCB achten Sie darauf, auch das letzte ‹|› zu entfernen. Endet Ihr regulärer Ausdruck mit ‹||› oder ‹|)›, werden auch leere Strings als gültige Kartennummer akzeptiert.

Um zum Beispiel nur Visa, MasterCard und American Express zu akzeptieren, nutzen Sie:

```
^(?:
4[0-9]{12}(?:[0-9]{3})? |         # Visa
5[1-5][0-9]{14} |                 # MasterCard
3[47][0-9]{13}                    # AMEX
)$
```

Regex-Optionen: Freiform
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Oder alternativ:

```
^(?:4[0-9]{12}(?:[0-9]{3})?|5[1-5][0-9]{14}|3[47][0-9]{13})$
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Suchen Sie in einem längeren Text nach Kreditkartennummern, ersetzen Sie die Anker durch Wortgrenzen (‹\b›).

Einbau der Lösung in eine Webseite

Das Beispiel in „Beispiel-Webseite mit JavaScript“ auf Seite 290 zeigt, wie Sie diese beiden regulären Ausdrücke in Ihr Bestellformular einbauen können. Das Eingabefeld für die Kreditkartennummer hat einen Event-Handler `onkeyup`, der die Funktion `validatecardnumber()` aufruft. Diese Funktion liest die Kartennummer aus dem Eingabefeld aus, entfernt Leerzeichen und Bindestriche und führt dann mithilfe des regulären Ausdrucks mit nummerierten einfangenden Gruppen eine Validierung durch. Das Ergebnis dieser Überprüfung wird angezeigt, indem der Text im letzten Absatz auf der Seite ersetzt wird.

Hat der reguläre Ausdruck keinen Erfolg, liefert `regexp.exec()` den Wert null zurück, und es wird `(Ungültige Kartennummer)` angezeigt. Passt die Regex, liefert `regexp.exec()` ein String-Array zurück. Das nullte Element dieses Arrays enthält das vollständige Suchergebnis. In den Elementen 1 bis 6 finden sich die Ergebnisse der sechs einfangenden Gruppen.

Unser regulärer Ausdruck hat sechs einfangende Gruppen, die in den Alternativen einer Alternation untergebracht sind. Das bedeutet, dass immer nur genau eine Gruppe an der Übereinstimmung beteiligt ist und die Kartennummer enthält. Die anderen Gruppen sind dann leer (entweder `undefined`, oder sie enthalten einen leeren String – das hängt von Ihrem Browser ab). Die Funktion prüft nacheinander die sechs einfangenden Gruppen. Findet sie eine, die nicht leer ist, wird die Kartenfirma erkannt und ausgegeben.

Zusätzliche Validierung mit dem Luhn-Algorithmus

Es gibt eine zusätzliche Validierungsmöglichkeit für Kreditkartennummern, bevor die Bestellung wirklich durchgeführt wird. Die letzte Ziffer in der Kartennummer ist eine Prüfsumme, die nach dem *Luhn-Algorithmus* berechnet wird. Da für diesen Algorithmus ein paar (wenn auch einfache) Kalkulationen notwendig sind, können Sie ihn nicht mit einem regulären Ausdruck implementieren.

Sie können Ihr Webseitenbeispiel für dieses Rezept mit dem Luhn-Algorithmus ergänzen, indem Sie vor der `else`-Zeile in der Funktion `validatecardnumber()` den Aufruf `luhn(cardnumber);` einfügen. So wird die Luhn-Prüfung nur dann durchgeführt, wenn der reguläre Ausdruck eine Übereinstimmung gefunden hat und nachdem die Kartenart ermittelt wurde. Allerdings ist das Bestimmen der Kartenart für die Luhn-Prüfung nicht notwendig. Alle Kreditkarten nutzen die gleiche Methode.

In JavaScript können Sie den Luhn-Algorithmus wie folgt implementieren:

```
function luhn(cardnumber) {
  // Aufbau eines Arrays mit den Ziffern der Kartennummer
  var getdigits = /\d/g;
  var digits = [];
  while (match = getdigits.exec(cardnumber)) {
    digits.push(parseInt(match[0], 10));
  }
  // Luhn-Algorithmus für die Ziffern ausführen
  var sum = 0;
  var alt = false;
  for (var i = digits.length - 1; i >= 0; i--) {
    if (alt) {
      digits[i] *= 2;
      if (digits[i] > 9) {
        digits[i] -= 9;
      }
    }
    sum += digits[i];
    alt = !alt;
  }
  // Prüfung der Kartennummer
  if (sum % 10 == 0) {
    document.getElementById("notice").innerHTML += '; Luhn-Prüfung erfolgreich';
  } else {
    document.getElementById("notice").innerHTML += '; Luhn-Prüfung nicht erfolgreich';
  }
}
```

Diese Funktion übernimmt einen String mit der Kreditkartennummer als Parameter. Die Kartennummer sollte nur aus Ziffern bestehen. In unserem Beispiel hat `validatecardnumber()` schon Leerzeichen und Bindestriche entfernt und ermittelt, ob die Kartennummer die richtige Anzahl an Ziffern besitzt.

In der Funktion wird zunächst der reguläre Ausdruck `‹\d›` verwendet, um über alle Ziffern im String iterieren zu können. Beachten Sie dabei den Modifikator `/g`. Innerhalb der Schleife findet sich in `match[0]` die Ziffer. Da reguläre Ausdrücke nur mit Texten arbeiten, rufen wir `parseInt()` auf, um sicherzustellen, dass die Variable als Integer und nicht als String gespeichert wird. Tun wir das nicht, findet sich später in der Variablen `sum` eine String-Verkettung von Ziffern und nicht die aufsummierten Werte.

Der eigentliche Algorithmus läuft über das Array und berechnet eine Prüfsumme. Lässt sich diese Summe ohne Rest durch 10 teilen, ist die Kartennummer gültig.

4.20 Europäische Umsatzsteuer-Identifikationsnummern

Problem

Sie sollen ein Onlinebestellformular für eine Firma in der Europäischen Union erstellen.

Kauft eine für die Umsatzsteuer registrierte Firma (Ihr Kunde), die in einem EU-Land sitzt, von einem Verkäufer (Ihre Firma) in einem anderen EU-Land etwas, muss der Verkäufer nach EU-Steuerrecht keine Umsatzsteuer berechnen. Hat der Käufer keine Umsatzsteuer-Identifikationsnummer (USt-IdNr.) angegeben, muss der Verkäufer die Mehrwertsteuer berechnen und sie an das lokale Finanzamt abführen. Um das zu vermeiden, nutzt der Verkäufer die USt-IdNr. des Käufers als Beweis, dass keine Steuer fällig ist. Für den Verkäufer ist es also sehr wichtig, die USt-IdNr. des Käufers zu überprüfen, bevor er die Bestellung ohne Umsatzsteuer durchführt.

Die häufigste Ursache für ungültige Umsatzsteuer-Identifikationsnummern sind einfache Tippfehler vom Kunden. Um den Bestellprozess schneller und einfacher zu gestalten, sollten Sie die USt-IdNr. mit einer Regex überprüfen, während der Kunde das Bestellformular ausfüllt. Das lässt sich mit etwas JavaScript-Code auf Clientseite oder im CGI-Skript auf Ihrem Webserver erreichen. Passt die Nummer nicht zum regulären Ausdruck, kann der Kunde den Tippfehler direkt korrigieren.

Lösung

Leerzeichen, Bindestriche und Punkte entfernen

Lesen Sie die vom Kunden eingegebene USt-IdNr. aus und speichern Sie sie in einer Variablen. Bevor Sie die Gültigkeit der Nummer prüfen, ersetzen Sie die durch folgenden regulären Ausdruck gefundenen Übereinstimmungen mit einem leeren String:

```
[-."]
```

Regex-Optionen: Keine
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

In Rezept 3.14 wird gezeigt, wie Sie diese Ersetzung durchführen können. Wir sind davon ausgegangen, dass der Kunde abgesehen von Bindestrichen, Punkten und Leerzeichen keine anderen Satzzeichen eingegeben hat. Jegliches weitere „falsche" Zeichen wird von der nächsten Prüfung abgefangen.

Überprüfen der Nummer

Nachdem Leerzeichen, Punkte und Bindestriche entfernt wurden, prüft dieser reguläre Ausdruck, ob die USt-IdNr. für einen der 27 EU-Staaten gültig ist:

```
^(
(AT)?U[0-9]{8} |                              # Österreich
(BE)?0?[0-9]{9} |                             # Belgien
(BG)?[0-9]{9,10} |                            # Bulgarien
(CY)?[0-9]{8}L |                              # Zypern
(CZ)?[0-9]{8,10} |                            # Tschechische Republik
(DE)?[0-9]{9} |                               # Deutschland
(DK)?[0-9]{8} |                               # Dänemark
(EE)?[0-9]{9} |                               # Estland
(EL|GR)?[0-9]{9} |                            # Griechenland
(ES)?[0-9A-Z][0-9]{7}[0-9A-Z] |               # Spanien
(FI)?[0-9]{8} |                               # Finnland
(FR)?[0-9A-Z]{2}[0-9]{9} |                    # Frankreich
(GB)?([0-9]{9}([0-9]{3})?|[A-Z]{2}[0-9]{3}) | # Großbritannien
(HU)?[0-9]{8} |                               # Ungarn
(IE)?[0-9]S[0-9]{5}L |                        # Irland
(IT)?[0-9]{11} |                              # Italien
(LT)?([0-9]{9}|[0-9]{12}) |                   # Litauen
(LU)?[0-9]{8} |                               # Luxemburg
(LV)?[0-9]{11} |                              # Lettland
(MT)?[0-9]{8} |                               # Malta
(NL)?[0-9]{9}B[0-9]{2} |                      # Niederlande
(PL)?[0-9]{10} |                              # Polen
(PT)?[0-9]{9} |                               # Portugal
(RO)?[0-9]{2,10} |                            # Rumänien
(SE)?[0-9]{12} |                              # Schweden
(SI)?[0-9]{8} |                               # Slowenien
(SK)?[0-9]{10}                                # Slowakei
)$
```

Regex-Optionen: Freiform, Groß-/Kleinschreibung wird ignoriert
Regex-Varianten: .NET, Java, PCRE, Perl, Python, Ruby

Dieser reguläre Ausdruck verwendet den Freiform-Modus, um ein späteres Bearbeiten der Regex zu vereinfachen. Schließlich nimmt die EU gelegentlich auch neue Länder auf, oder die Staaten passen ihre Regeln für die USt-IdNr. an. Leider ist in JavaScript kein Freiform-Modus möglich. Dort müssen Sie alles in einer Zeile unterbringen:

```
^((AT)?U[0-9]{8}|(BE)?0?[0-9]{9}|(BG)?[0-9]{9,10}|(CY)?[0-9]{8}L|": "
(CZ)?[0-9]{8,10}|(DE)?[0-9]{9}|(DK)?[0-9]{8}|(EE)?[0-9]{9}|": "
(EL|GR)?[0-9]{9}|(ES)?[0-9A-Z][0-9]{7}[0-9A-Z]|(FI)?[0-9]{8}|": "
(FR)?[0-9A-Z]{2}[0-9]{9}|(GB)?([0-9]{9}([0-9]{3})?|[A-Z]{2}[0-9]{3})|": "
(HU)?[0-9]{8}|(IE)?[0-9]S[0-9]{5}L|(IT)?[0-9]{11}|": "
(LT)?([0-9]{9}|[0-9]{12})|(LU)?[0-9]{8}|(LV)?[0-9]{11}|(MT)?[0-9]{8}|": "
(NL)?[0-9]{9}B[0-9]{2}|(PL)?[0-9]{10}|(PT)?[0-9]{9}|(RO)?[0-9]{2,10}|": "
(SE)?[0-9]{12}|(SI)?[0-9]{8}|(SK)?[0-9]{10})$
```

Regex-Optionen: Groß-/Kleinschreibung wird ignoriert
Regex-Varianten: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

In Rezept 3.6 wird beschrieben, wie Sie diesen regulären Ausdruck auf Ihrem Bestellformular unterbringen können.

Diskussion

Leerzeichen, Punkte und Bindestriche entfernen

Damit die Umsatzsteuer-Identifikationsnummern für Menschen leichter lesbar sind, werden sie häufig mit zusätzlichen Trennzeichen eingegeben. So könnte ein deutscher Kunde seine USt-IdNr. `DE123456789` zum Beispiel als `DE 123.456.789` eingeben.

Ein einzelner regulärer Ausdruck, der die Nummern aus 27 Ländern in allen möglichen Schreibweisen erkennt, ist unmöglich zu realisieren. Da die Trennzeichen nur der Lesbarkeit dienen, ist es viel einfacher, zunächst alle diese Zeichen zu entfernen und dann die reine USt-IdNr. zu analysieren.

Der reguläre Ausdruck `‹[-.”]›` passt zu einem Zeichen, das ein Bindestrich, ein Punkt oder ein Leerzeichen ist. Ersetzt man alle Übereinstimmungen dieses regulären Ausdrucks durch einen leeren String, werden diese Zeichen aus den Nummern entfernt.

Umsatzsteuer-Identifikationsnummern bestehen nur aus Buchstaben und Ziffern. Statt lediglich die am häufigsten eingegebenen Trennzeichen mit `‹[-.”]›` zu entfernen, können Sie auch alle ungültigen Zeichen mit `‹[^A-Z0-9]›` eliminieren.

Validieren der Nummer

Die zwei regulären Ausdrücke für das Validieren der Nummer sind identisch. Nur nutzt der erste den Freiform-Modus, damit er leichter lesbar ist. JavaScript unterstützt diesen Modus nicht, aber bei den anderen Varianten haben die Sie freie Wahl.

Die Regex nutzt eine große Alternation, um die Umsatzsteuer-Identifikationsnummern aller 27 EU-Staaten berücksichtigen zu können. Die Formate sehen so aus:

Belgien
: *999999999* oder 0*999999999*

Bulgarien
: *999999999* oder *9999999999*

Dänemark
99999999

Deutschland
999999999

Estland
999999999

Finnland
99999999

Frankreich
XX*999999999*

Griechenland
999999999

Großbritannien
999999999, *999999999999* oder XX*999*

Irland
9S*99999*L

Italien
99999999999

Lettland
99999999999

Litauen
999999999 oder *99999999999*

Luxemburg
99999999

Malta
99999999

Niederlande
*999999999*B*99*

Österreich
U*99999999*

Polen
999999999

Portugal
999999999

Rumänien
99, *999*, *9999*, *99999*, *999999*, *9999999*, *99999999*, *999999999* oder *9999999999*

Schweden
99999999999

Slowakei
999999999

Slowenien
: *99999999*

Spanien
: X*9999999*X

Tschechische Republik
: *99999999*, *999999999* oder *9999999999*

Ungarn
: *99999999*

Zypern
: *99999999*L

Streng genommen ist der zweistellige Ländercode Teil der USt-IdNr. Aber viele lassen ihn häufig weg, da die Rechnungsanschrift schon den Staat angibt. Der reguläre Ausdruck akzeptiert daher die Nummern mit und ohne Ländercode. Wenn Sie wollen, dass er eingegeben werden muss, entfernen Sie alle Fragezeichen aus dem regulären Ausdruck. Dann sollten Sie aber auch in der Fehlermeldung, die den Anwender auf eine ungültige USt-IdNr. hinweist, erwähnen, dass der Ländercode eingegeben werden muss.

Akzeptieren Sie Bestellungen nur aus bestimmten Ländern, können Sie die Länder aus der Regex weglassen, die in der Länderauswahl auf Ihrem Bestellformular vorhanden sind. Löschen Sie eine der Alternativen, müssen Sie auch den Operator ‹|› löschen, der die Alternativen untereinander trennt. Tun Sie das nicht, steht in Ihrem regulären Ausdruck ‹||›. Das führt aber zu einer Alternative, die einen leeren String akzeptiert, sodass Ihr Bestellformular letztendlich auch ohne USt-IdNr. fertiggestellt werden kann.

Die 27 Alternativen sind in einer Gruppe zusammengefasst. Diese Gruppe umfasst den gesamten Bereich zwischen einem Zirkumflex und einem Dollar, wodurch der reguläre Ausdruck mit Anfang und Ende des zu überprüfenden Strings verbunden ist. Die gesamte Eingabe muss also eine gültige USt-IdNr. sein.

Suchen Sie in einem längeren Text nach Umsatzsteuer-Identifikationsnummern, ersetzen Sie die Anker durch die Wortgrenzen ‹\b›.

Variationen

Der Vorteil eines regulären Ausdrucks für alle 27 Staaten liegt darin, dass Sie in Ihrem Formular nur eine Regex-Überprüfung benötigen. Sie könnten aber auch 27 getrennte Regexes nutzen. Zuerst prüfen Sie das Land, das der Kunde in der Rechnungsanschrift angegeben hat. Dann validieren Sie die USt-IdNr. abhängig vom Land:

Belgien
: ‹^(BE)?0?[0-9]{9}$›

Bulgarien
: ‹^(BG)?[0-9]{9,10}$›

Dänemark
: ‹^(DK)?[0-9]{8}$›

Deutschland

```
‹^(DE)?[0-9]{9}$›
```

Estland

```
‹^(EE)?[0-9]{9}$›
```

Finnland

```
‹^(FI)?[0-9]{8}$›
```

Frankreich

```
‹^(FR)?[0-9A-Z]{2}[0-9]{9}$›
```

Griechenland

```
‹^(EL|GR)?[0-9]{9}$›
```

Großbritannien

```
‹^(GB)?([0-9]{9}([0-9]{3})?|[A-Z]{2}[0-9]{3})$›
```

Irland

```
‹^(IE)?[0-9]S[0-9]{5}L$›
```

Italien

```
‹^(IT)?[0-9]{11}$›
```

Lettland

```
‹^(LV)?[0-9]{11}$›
```

Litauen

```
‹^(LT)?([0-9]{9}|[0-9]{12})$›
```

Luxemburg

```
‹^(LU)?[0-9]{8}$›
```

Malta

```
‹^(MT)?[0-9]{8}$›
```

Niederlande

```
‹^(NL)?[0-9]{9}B[0-9]{2}$›
```

Österreich

```
‹^(AT)?U[0-9]{8}$›
```

Polen

```
‹^(PL)?[0-9]{10}$›
```

Portugal

```
‹^(PT)?[0-9]{9}$›
```

Rumänien

```
‹^(RO)?[0-9]{2,10}$›
```

Schweden

```
‹^(SE)?[0-9]{12}$›
```

Slowakei

```
‹^(SK)?[0-9]{10}$›
```

Slowenien

```
‹^(SI)?[0-9]{8}$›
```

Spanien

```
‹^(ES)?[0-9A-Z][0-9]{7}[0-9A-Z]$›
```

Tschechische Republik

```
‹^(CZ)?[0-9]{8,10}$›
```

Ungarn

```
‹^(HU)?[0-9]{8}$›
```

Zypern

```
‹^(CY)?[0-9]{8}L$›
```

Implementieren Sie Rezept 3.6, um die USt-IdNr. mit der ausgewählten Regex zu validieren. Damit erfahren Sie, ob die Nummer für das Land, das der Kunde angegeben hat gültig ist.

Der Hauptvorteil der getrennten regulären Ausdrücke ist, dass die USt-IdNr. auf jeden Fall mit der korrekten Länderkennung beginnen kann, ohne dass der Kunde sie angeben muss. Passt der reguläre Ausdruck auf die angegebene Nummer, prüfen Sie den Inhalt der ersten einfangenden Gruppe. In Rezept 3.9 wird beschrieben, wie Sie das machen können. Ist die erste einfangende Gruppe leer, hat der Kunde den Ländercode nicht mit eingegeben. Sie können ihn dann selbst ergänzen, bevor Sie die überprüfte Nummer in Ihrer Bestelldatenbank ablegen.

Griechische Umsatzsteuer-Identifikationsnummern können zwei verschiedene Ländercodes haben. `EL` wird traditionell für griechische Nummern verwendet, aber `GR` ist der ISO-Ländercode für Griechenland.

Siehe auch

Der reguläre Ausdruck prüft nur, ob die Nummer wie eine gültige USt-IdNr. aussieht. Das reicht aus, um echte Fehler auszuschließen. Aber ein regulärer Ausdruck kann offensichtlich nicht prüfen, ob die Nummer der Firma zugeordnet ist, die die Bestellung aufgibt. Die Europäische Union hat eine Website (*http://ec.europa.eu/taxation_customs/vies/vieshome.do*), auf der Sie prüfen können, zu welcher Firma eine bestimmte USt-IdNr. gehört – wenn sie denn überhaupt zugewiesen ist. Die Nummern werden mit der Datenbank des entsprechenden Staats verglichen. Manche Staaten bestätigen dabei allerdings nur eine Gültigkeit, ohne weitere Informationen über die entsprechende Firma herauszugeben.

Die in diesem regulären Ausdruck genutzten Techniken werden in den Rezepten 2.3, 2.5 und 2.8 besprochen.