PAYMENT REQUEST API

Zahlen, bitte!

Das Bezahlen im Browser soll vereinheitlicht werden. Noch gibt es einige Haken.

Wenn sich Google, Mozilla, Microsoft und Facebook unter dem Dach des W3C zusammentun, um am Thema Online-Payment zu arbeiten, ist das automatisch schon spannend. Der Name Payment Request API lässt die Hoffnung auf etwas Großes ansteigen. Erfährt man dann bei genauerer Betrachtung, dass es lediglich um die Vereinheitlichung des Zahlungsprozesses am Ende eines Bestellvorgangs geht, weicht der anfängliche Optimismus ein wenig. Klar, wenn weltweit alle Zahlungsvorgänge dieselben wären, würde das den Konversionsraten und dem Kundenvertrauen guttun, aber wie sieht es da mit der User Experience (UX) aus? Passt dieser Prozess wirklich in jeden Shop, oder wirkt er wie ein vom Browser vorgegebener Fremdkörper? Ein kurzes Für und Wider wird weiter unten diskutiert, zunächst einmal geht es um die nackten technischen Tatsachen.

Rahmenbedingungen

Zum Redaktionsschluss hat das Spezifikationsdokument den Status einer W3C Candidate Recommendation, befindet sich also nur noch eine Stufe vor einem endgültigen Standard [1]. Zuletzt aktualisiert wurde das Ganze im Dezember 2019, wenngleich es auch wie üblich einen „Editor's Draft" gibt [2], wo Spöttern zufolge jeder korrigierte Kommafehler das Änderungsdatum hochsetzt. Hier gibt es quasi im Monatsrhythmus etwas Neues.

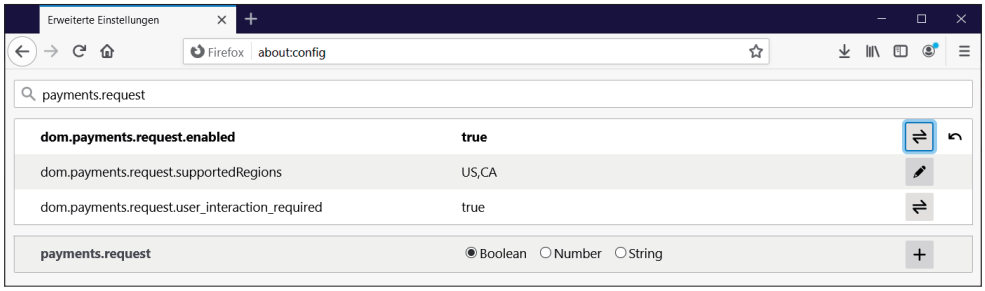
Hinsichtlich der Browserunterstützung gibt es Licht und Schatten zu vermelden [3]: Chrome-Browser und ihre Geschwister mit derselben Engine (Edge, Opera) unterstützen die Spezifikation komplett; Safari ist – inklusive der Variante für iOS und iPadOS – auch mit an Bord, unterstützt allerdings nur Apple Pay, weswegen dieser Browser hier vorerst außen vor bleibt. Selbst die jüngsten Versionen von Edge mit Microsoft-Engine bieten API-Support, im Gegensatz zum Vorgänger, dem Internet Explorer. Die beiden letztgenannten Browser haben allerdings ohnehin einen rapide sinkenden Marktanteil und werden nicht mehr weiterentwickelt. Der Microsoft-Vertreter im Autorenkreis des Standards ist inzwischen folgerichtig ausgeschieden.

Etwas komplizierter ist die Lage beim Browser Firefox. Unter Android wird das Payment Request API gar nicht unterstützt; Nutzer der Desktop-Variante müssen das Feature erst aktivieren. Unter dem „geheimen" URL *about:config* muss erst eine Sicherheitswarnung weggeklickt und dann die Einstellung *dom.payments.request.enabled* auf *true* gesetzt werden (Bild 1). Eine weitere relevante Einstellung ist *dom.payments.request.supportedRegions* mit dem Standardwert „US,CA" (steht für USA, Kalifornien). Dies soll es ermöglichen, auch außerhalb dieses US-Bundesstaats das Payment Request API zu verwenden. Die Erfolgsquote ist arg durchwachsen: Auf verschiedenen Testsystemen ließen sich weder der normale Firefox noch die Nightly-Version zur Kooperation bewegen. In vereinzelten Ländern gibt es Berichte, man müsse den Code seines eigenen Landes mit angeben (also etwa: *DE*, *AT* oder *CH*), doch auch dies führte nicht immer zum Ziel. In der Konsole der F12-Browser-Tools können Sie den Erfolg sofort überprüfen: *window.PaymentRequest* darf nicht *undefined* sein. Dieser Artikel geht in seinem praktischen Teil den Weg des geringsten Widerstands und setzt ausschließlich auf Google Chrome.

Da es um Zahlungsdaten geht, steht das API ausschließlich über eine sichere Verbindung (HTTPS) zur Verfügung; bei Verwendung von *localhost* funktionieren auch selbstsignierte Zertifikate. Andernfalls erscheint teilweise nicht einmal eine Fehlermeldung in der Konsole.

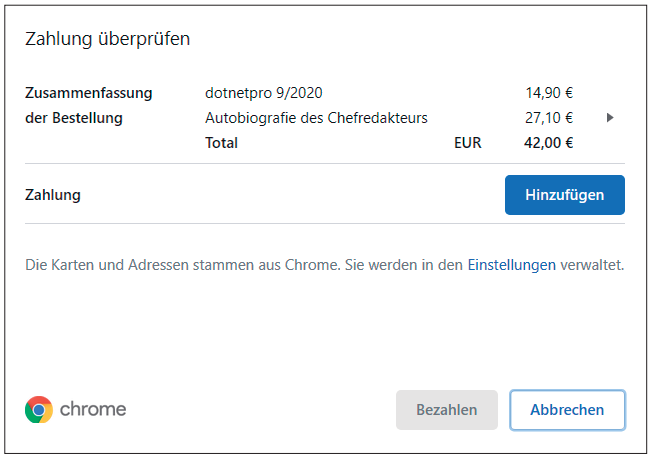
Konfiguration

Das Hauptobjekt des API heißt, nicht überraschend, *PaymentRequest*. Eine Zahlungsanforderung erwartet zwei Konfigurationsobjekte: Informationen über die erlaubten Zahlungsmethoden, und Daten über das, was gezahlt werden soll. Bei den Zahlungsmethoden ist anzugeben, welche Zahlungsnetzwerke (etwa Visa, Mastercard) und welche Arten von Zahlungskarten (beispielsweise Kreditkarte, Debitkarte) unterstützt werden sollen. Hier ein Beispiel in objektliteraler Syntax:

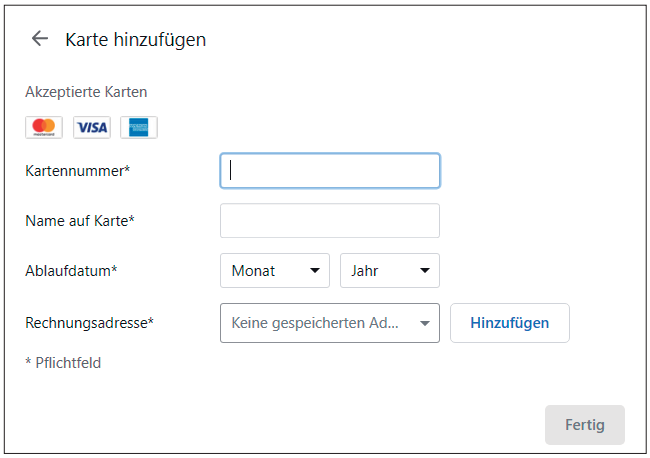


Payment Request API in Firefox aktivieren (Bild 1)

```
var methods = [{
    supportedMethods:
    "basic-card",
    data: {
      supportedNetworks:
      ["mastercard",
```



Bezahlen im Browser ... (Bild 2)



... mit Zugangsdaten, die sich im Browser speichern lassen (Bild 3)

```
      "visa", "amex"], supportedTypes: ["debit",
      "credit"]}
}];
```

Das zweite Konfigurationsobjekt ist im Wesentlichen der Warenkorb. Neben einer ID (die jedoch nicht immer vorliegt) sind zwei weitere Werte entscheidend: *displayItems* enthält die zu bezahlenden Artikel, *total* die für die Zahlung relevanten Daten, nämlich Gesamtsumme und Währung. Das kann beispielsweise wie folgt aussehen:

```
var data = { id: "123",
  total: { label: "Gesamtbetrag",
    amount: {
    currency: "EUR", value: "42" }
  },
  displayItems: [{
    label: "dotnetpro 9/2020",
    amount: {
    currency: "EUR", value: "14.90" }
  },
  {
    label: "Autobiografie des Chefredakteurs",
    amount: { currency: "EUR", value: "27.10" }
  }]
};
```

Das Payment Request API addiert die einzelnen Elemente nicht zusammen; der Wert unter *total/amount* muss also korrekt sein. Bei mehreren verschiedenen Versandoptionen können diese im Schlüssel *shippingOptions* in den Daten angegeben werden; sie erscheinen dann in der Browser-Oberfläche als Auswahlmöglichkeiten.

Anzeige

Mit diesen Daten bewaffnet lässt sich ein *PaymentRequest*-Objekt erstellen und anzeigen. Zunächst wird das Objekt mit den Konfigurationsoptionen bestückt:

```
var paymentRequest = new PaymentRequest(methods, data);
```

Im optionalen dritten Parameter können zusätzliche Daten abgefragt werden, die für die Zahlung beziehungsweise Bestellung notwendig sind, etwa die Telefonnummer. Der Wert des Parameters kann wie folgt aussehen:

```
var options = {
  requestPayerName: true,
  requestPayerEmail: true,
  requestPayerPhone: true,
  requestShipping: true
};
```

Als Nächstes bietet es sich an, zu prüfen, ob diese Zahlungsanfrage vom aktuellen Browser überhaupt durchgeführt werden kann (etwa: korrektes Zahlungsnetzwerk – Safari unterstützt beispielsweise nur Apple Pay). Dies erledigt die Methode *canMakePayment()*, die anstelle eines erwartbaren booleschen Werts ein *Promise* zurückliefert:

```
paymentRequest.canMakePayment().then(
  function (result) {
    if (result) {
      // UI anzeigen
    }
});
```

Dann instruiert die Methode *show()* den Browser, den Zahlvorgang zu starten, indem die Daten angezeigt werden.

```
paymentRequest.show();
```

Aussehen und UI übernimmt hierbei komplett der Browser. Bild 2 zeigt das UI in Google Chrome; Nutzer können auch Zahlungsdaten hinterlegen, die dann im Browser (und nicht im Online-Shop) gespeichert werden, wie in Bild 3 zu sehen. Die Auswahlmöglichkeiten bei den Zahlungsmöglichkeiten korrespondieren dabei mit den Vorgaben im ersten Parameter für *PaymentRequest*. Würde etwa der Wert *„amex“* weggelassen werden, dann würde auch das blaue American-Express-Logo aus Bild 3 verschwinden. ▶

Listing 1: Google Pay und Apple Pay via Payment Request API (Auszug)

```
var basicCardMethod = {
  supportedMethods: "basic-card",
  data: {
    supportedNetworks:
    ["mastercard", "visa", "amex"],
    supportedTypes:
    ["debit", "credit"] }
};
var googlePayMethod = {
  supportedMethods: "https://google.com/pay",
  data: {
    apiVersion: 2,
    merchantInfo:
    {merchantName: "Webshop XYZ" },
    allowedPaymentMethods: [{
      type: "CARD", parameters: {
      allowedAuthMethods:
      ["CRYPTOGRAM_3DS", "PAN_ONLY"],
      allowedCardNetworks:
      ["mastercard", "visa", "amex"]}
    }]
  }
};
var applePayMethod = {
  supportedMethods: "https://apple.com/apple-pay",
  data: {
    apiVersion: 3,
    merchantIdentifier:
    "merchant.com.example",
    merchantCapabilities:
    ["supportsCredit", "supportsDebit",
      "supports3DS"],
    supportedNetworks:
    ["masterCard", "visa", "amex"],
    countryCode: "DE" }
};
var data = { /* ... */ };
var paymentRequest = new PaymentRequest(
  [basicCardMethod, googlePayMethod, applePayMethod],
  data);
```

Chrome speichert Kreditkarten auch im Google-Profil, was den Vorteil der einfachen Verteilung auf mehrere Systeme hat, und natürlich den Nachteil, dass Google die Daten dann auch hat. Wer gerne im Inkognito-Modus testet, muss sich hier umgewöhnen: Die Schaltfläche zum Speichern von Kreditkartendaten ist zwar aktiv, aber nur im normalen Browser-Modus funktional.

Zahlungsvorgang

Die Methode *show()* liefert ein *Promise* zurück, denn irgendwann wird der Zahlprozess abgeschlossen werden. Der Browser wartet dann auf die Ausführung der *complete()*-Methode. Dazwischen ist aber die eigene Implementierung dran. Die erhält nämlich alle vom Nutzer angegebenen Daten zur Zahlung wie Kreditkarteninfos oder Rechnungsadresse. Mit diesen lässt sich die Zahlung durchführen, entweder selbst oder durch Einschalten eines Zahlungsdienstes. Der folgende Code überspringt dies und führt sofort *complete()* aus; die Variable *response* enthält die angesprochenen Zahlungsinfos (Bild 4 zeigt einen Auszug davon):

```
paymentRequest.show().then(
  function(response) {
    //TODO: die Zahlungsdaten verarbeiten
    response.complete("success");
});
```

Hierbei kann durchaus ein Fehler auftreten, etwa wenn Nutzer das Zahlungsfenster mit der Taste [Esc] schließen. Ein Abfangen des Fehlers tut also Not (wenngleich das im Rahmen des Beispiels nur pro forma ergänzt werden soll):

```
.catch((error)=> {
    //TODO: ausführliche Fehlerbehandlung
    console.error(error);
});
```

Mobiles Payment

Gerade auf mobilen Endgeräten ist die Verwendung von zentral gespeicherten Zahlungsdaten interessant. Bei Android und iOS kommen hier natürlich sofort Google Pay und Apple Pay ins Spiel.

Der Chrome-Browser unterstützt Google Pay, allerdings nur mobil, nicht auf dem Desktop. Hierzu müssen außerdem zusätzliche Daten bei der Initialisierung von *PaymentRequest* angegeben werden (siehe Listing 1). Darüber hinaus ist eine Registrierung bei Google als *Merchant* erforderlich [5].

```
▼PaymentResponse
 ▼details:
  ▼billingAddress:
   ▶addressLine: ["Spreeweg 1"]
    city: "Berlin"
    country: "DE"
    dependentLocality: ""
    organization: ""
    phone: "+493000000000"
    postalCode: "10557"
    recipient: "Christian Wenz"
    region: ""
    sortingCode: ""
   ▶__proto__: Object
  cardNumber: "378282246310005"
  cardSecurityCode: "1234"
  cardholderName: "Christian Wenz"
  expiryMonth: "06"
  expiryYear: "2022"
 ▶__proto__: Object
 methodName: "basic-card"
```

Auszug der Zahlungsdaten, auf die JavaScript Zugriff erhält (Bild 4)

Bei Apple Pay sieht es ähnlich aus: Unter Safari ist das aktuell die einzige unterstützte Zahlungsmethode für *PaymentRequest*, auch hier sind spezifische Zusatzangaben sowie eine Registrierung [6] notwendig. Früher gab es für Apple Pay nur eine Apple-eigene JavaScript-Bibliothek zum Auslösen von Zahlungen,

inzwischen wird *PaymentRequest* direkt unterstützt. Der Anstoß dafür kam möglicherweise von der Konkurrenz: Das Open-Source-Projekt *appr-wrapper* von Google [6] hat vorgemacht, wie man Apple Pay ansprechen kann.

Diskussion

Ein sicherer, bequemer und vertrauenswürdiger Bezahlvorgang ist das A und O im E-Commerce. Insofern ist das Payment Request API eine spannende Ergänzung. Auch andere Zahlungsanbieter wie etwa Stripe bieten in ihrem API eine Unterstützung von *PaymentRequest* [4] (und einen Fallback für nicht unterstützte Browser).

Apropos Browser: Der unvollständige Support (konkret: die nicht standardmäßige Aktivierung in Firefox) erschwert den Einsatz ohne Rückfallmöglichkeit auf eine Alternative. Dass die Zahlungsdaten dann direkt im Browser, gegebenenfalls samt zugehöriger Cloud landen, ist nicht unbedingt wünschenswert. Bei vielen, gerade kleineren Webshops landen die Kreditkartendaten überhaupt nicht beim Site-Betreiber, sondern es wird über einen eigenen Zahlungsdienstleister abgerechnet. Dieser ist dann der einzige, der die Informationen erhält. Beim Payment Request API landen diese, wie in Bild 4 zu sehen, zunächst im JavaScript-Code. Und nicht jede Webanwendung möchte den bei Nutzern eher unbekannten Weg über den browsereigenen Zahlungsprozess gehen, sondern in seinem Design und Workflow bleiben. Es ist abzuwarten, ob eine finale Version des Standards sowie eine breitere Verfügbarkeit in Browsern die Verwendung des Features ankurbeln können. ■

[1] W3C Candidate Recommendation, www.dotnetpro.de/SL2009BrowserAPI1
[2] W3C Editors Draft, www.dotnetpro.de/SL2009BrowserAPI2
[3] Can I use, Payment Request API, www.dotnetpro.de/SL2009BrowserAPI3
[4] Stripe Docs, Payment Request Button, www.dotnetpro.de/SL2009BrowserAPI4
[5] Pay Google, www.dotnetpro.de/SL2009BrowserAPI5
[6] Configure Apple Pay, www.dotnetpro.de/SL2009BrowserAPI6

Christian Wenz
ist Mitgründer der Agentur Arrabiata Solutions GmbH (www.arrabiata.de) und verantwortet dort die Themen Performance, mobile Anwendungen und Security. Er ist MVP für ASP.NET und ASPInsider.

dnpCode A2009BrowserAPI