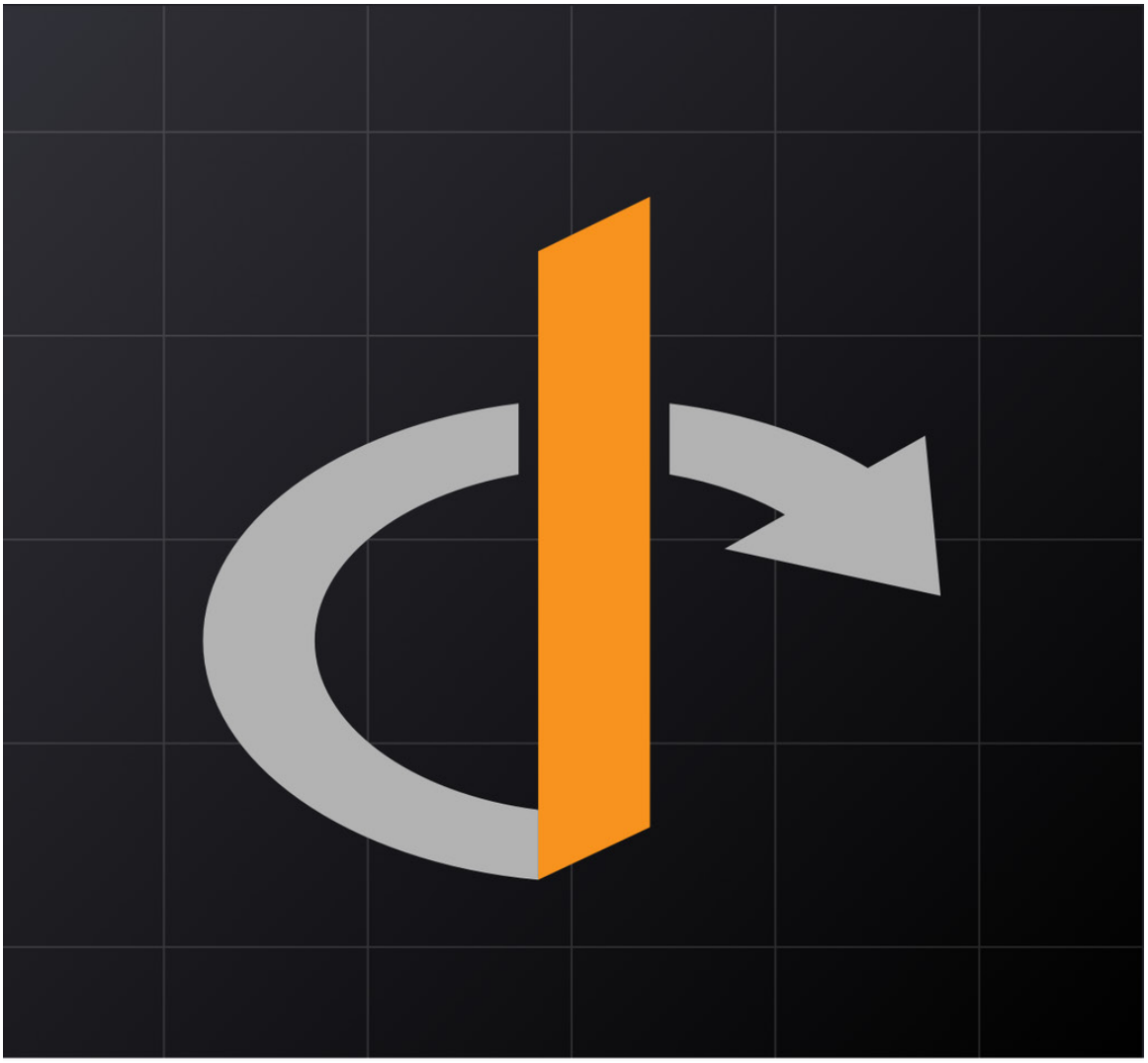


THE OPENID CONNECT ハンドブック

By Bruno Krebs

Auth0

目次

はじめに **4**

エンティティとアイデンティティについて 5

認証と認可 6

まとめ 8

OpenID Connectの概要 **9**

アプリケーションプロトコルの概説 10

OpenID Connectの仕組み 13

OAuth 2.0の概要 15

OpenID Connectのユースケース 17

まとめ 18

OpenID Connectの実行 **20**

Auth0アカウントの作成 21

前提条件 22

OpenID Connectと従来のウェブアプリケーション **24**

OpenID Connectと認証フロー 25

プロジェクトのブートストラップ 25

通常のウェブアプリケーションをAuth0に登録する 26

Discovery Endpointからの情報取得 28

認証プロセスの開始 30

認証コールバックへの対処 35

ユーザーに関する詳細情報の要求 40

ユーザー認証にSDKを使用する 41

まとめ 46

従来のウェブアプリケーションと委任認可フロー **47**

認可コードフロー 48

リソースサーバーを使用する 49

委任認可のリクエスト 53

委任認可のリクエストにSDKを使用 61

まとめ 64

OpenID Connectとシングルページアプリケーション **65**

SPAの認証処理 65

非推奨のインプリシットグラントの代替手段について 66

シングルページアプリケーションをAuth0に登録 67

プロジェクトのブートストラップ 69

SPAとOIDCプロバイダを統合 71

SPAとOIDCプロバイダを統合 – 簡単な方法 78

まとめ 83

はじめに

本書では、エンドユーザーの認証処理とアイデンティティ検証のために、あらゆる種類のアプリケーションで使用できるプロトコル、OpenID Connectについて学習します。プロトコルの説明に進む前に、まず、認証、認可、エンティティ、アイデンティティなどのトピックを紹介します。これらのトピックは、OpenID Connectや関連するテクノロジについて学習する際の基礎となるものです。

次章では、OpenID Connectが必要とされる理由、類似プロトコルとの違い、そしてOpenID Connectを構成する要素について詳しく学びます。最後に、OpenID Connectの実務での応用について、通常のウェブアプリケーション、シングルページアプリケーション（SPA）、ネイティブアプリケーションなど、さまざまなアプリケーションアーキテクチャにおけるユーザーの認証とアイデンティティ確認を行う方法を参照しながら学んでいきます。

本書を一読することにより、OpenID Connectとは何か、このプロトコルをいつどのように活用できるか、そしてプロトコルがどのように機能するかを習得できます。この知識を土台にすることで、正しい情報に基づいてプロトコルを使用し、ニーズに合わせてプロトコルを拡張するほか、直面する可能性のある問題に対応できるようになります。

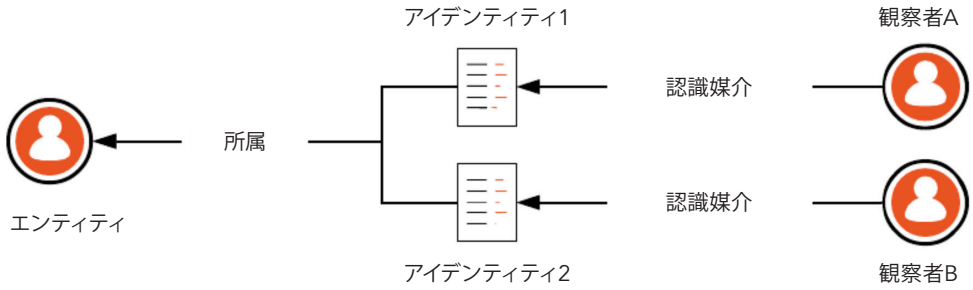
エンティティとアイデンティティについて

エンティティとアイデンティティという用語は、私たちの生活の中で頻繁に使用されていますが、誤解されていたり、誤った使い方をされたりすることも少なくありません。本書を読み進めるにあたり知っておく必要があることとして、**エンティティ**とは個々のユニットとして存在するものを指すのに対し、**アイデンティティ**とはコンテキスト内でエンティティを区別するために使用される属性のセットであるということです。この定義を聞いても、これら用語の意味の区別がつかない場合のために、2つほど例を挙げてご説明します。

まず、誰か一人の人物、たとえば科学者アインシュタインについて考えてみてください。最も基本的な定義によると、アインシュタインはエンティティです。つまり、何かを考えたり、話したり、歩いたり、教えたりすることのできるエンティティです。ただし、アインシュタインの周囲の人々は、彼をエンティティとして直接認識していたわけではなく、彼の名前や性別、身長、どこに住んでいるかなどのさまざまな属性を通じて、アインシュタインという人物を間接的に認識していました。

アインシュタインを認識するために人々が使用したさまざまな属性のサブセットは、アインシュタインのアイデンティティを形成しました。一部の人々の間では、アインシュタインは学歴のある名高い教授として認識されていましたが、銀行にとっては、アインシュタインは署名と口座番号を持つ顧客でした。

エンティティとアイデンティティが何を表し、どのように関連しているかについて理解するのに役立つもう1つの例は、ソフトウェアアプリケーションを分析することです。アインシュタインと同様に、アプリケーションも（周囲のものの認識とは無関係に）個々のユニットとして存在するエンティティです。また、エンティティとして、アプリケーションに複数のアイデンティティがあることも、アインシュタインと同じです。このアプリケーションを使用するSPAにとって、アプリケーションのアイデンティティは、インターネットドメイン、TLS証明書などで構成されます。データベースにとって、このアプリケーションのアイデンティティは、クレデンシャル（ユーザー名やパスワードなど）とそのアクセス許可（アプリケーションがどのテーブルにアクセスできるかなど）です。



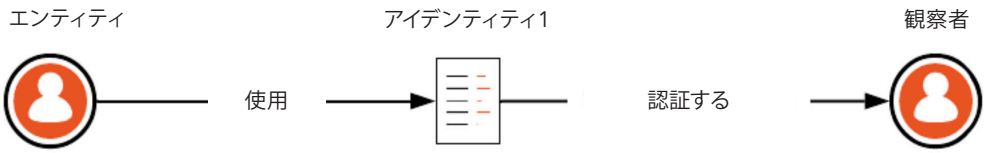
上の図は、2つの異なるアイデンティティを通じてエンティティを認識する、2人の異なる観察者を示しています。つまり、各観察者は、異なる属性（アイデンティティ）のセットを使用して、同じエンティティを理解しています。

このセクションで習得すべき最も重要なコンセプトは、これらのアイデンティティを形成する属性が、エンティティの置かれているコンテキストによって変わるということです。この定義は、OpenID Connectを使用してユーザーを識別する方法を学習する際に重要となります。

認証と認可

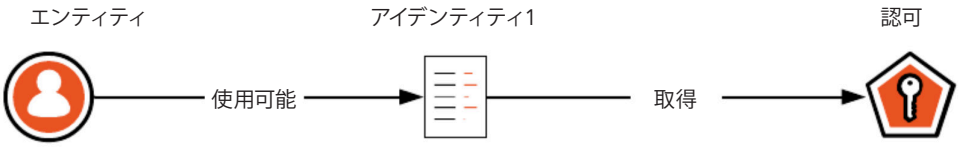
OpenID Connectの説明に進む前に取り上げておくべきトピックが、さらに2つあります。認証と認可です。前のセクションで取り上げた用語と同様に、この2つのトピックは大きな混乱を引き起こす可能性があり、誤用されていることも少なくありません。そのため、これらの用語の意味と関係性を理解しておくことが重要です。

まず、認証とは、エンティティ（ユーザーなど）のアイデンティティを確認するプロセスです。通常、認証プロセスは、何らかの形式の証明に依存しています。たとえば、銀行に行って口座から現金を引き出そうとすると、身元を確認するための身分証明書（公式文書）の提示を行員から求められることがあります。同様に、航空券を購入する場合も、その航空券を受け取る権利があることをパスポートで証明しないと、飛行機に搭乗できないことがあります。どちらの例も、身元確認のための認証プロセスが行われる実際の状況を示しています。



それとは対照的に、認可とは、エンティティが何にアクセスでき、どのアクションを実行できるかを検証するプロセスを指します。具体的な例として、演劇のチケットを購入する状況を思い浮かべてみてください。この場合、主催者側にとって重要なのは、チケット購入者のアイデンティティ（つまり身元）を確認することではなく、その演劇を観賞する権利が購入者にあるかどうかです。チケット購入者は、観劇する権利が自分にあることを証明するために、パスポートなどの身分証明書ではなく、自分に関する情報を含んでいないチケットを使用します。

上記の説明によって、認証と認可のトピックが多少解明されるかもしれませんが、これらの用語の定義と用法が互いに重複していることもよくあります。たとえば、銀行のシナリオをさらに深く考えてみると、口座名義人に口座残高へのアクセスを許可するために身元確認も行われていることに気付くでしょう。つまり、行員は、名義人以外の人が名義人の身分証明書を携帯しており、署名を複製する方法を知っていたとしても、現金の受け取りを許可したりしません。



ここで理解しておくべき重要なことは、認証は認可につながる可能性があるが、その逆はあり得ないということです。アイデンティティ（身元）を証明することは何かにアクセスする権利を得る（つまり、何かを行う認可を受ける）うえで十分かもしれませんが、（演劇のチケットを購入する例のように）認可を受けていることがエンティティの身元確認になるわけではありません。このシナリオでは、チケットを所持していても、身元（氏名や年齢など）の確認になるわけではありません。チケットが証明するのは、チケットの所持者に演劇を観賞する権利があるということだけです。

まとめ

本書の最初の章では、OpenID Connectを理解するための基礎となる、エンティティ、アイデンティティ、認証、認可などのトピックを紹介しました。要約すると、エンティティはどれも複数のアイデンティティを持つことができ（たとえば、アインシュタインは教授としても顧客としても認識される）、アイデンティティとは当該エンティティに属するプロパティのセットであることがわかりました。それに加えて、認証と認可およびそれらの関係について学びました。具体的には、エンティティがそのアイデンティティを使用して何らかのアクションを実行するための認可を受けることはできますが、その逆は不可能である（つまり、認可を受けたからといって、認証された、身元確認されたわけではない）ことを学びました。この知識を念頭に置いて、OpenID Connectプロトコルについて、そしてこのテクノロジを使用してエンドユーザー認証を処理する方法について学んでいきましょう。

OpenID Connectの概要

OpenID Connect（略してOIDC）プロトコルにより、さまざまなタイプのアプリケーションが、安全で一元化かつ標準化された方法で認証とアイデンティティ管理をサポートできるようになります。OpenID Connectプロトコルに基づくアプリケーションでは、認証プロセスの安全な処理とユーザーのアイデンティティ（つまり、個人属性）の検証が、アイデンティティプロバイダによって行われます。

* **重要な用語：**デジタルアイデンティコミュニティで広く使用されているにもかかわらず、仕様書には、OpenID Connectがアイデンティティプロバイダを採用しているとは明記されていません。OpenID Connectプロトコルでは、エンドユーザーの認証を担当するエンティティを指す用語として認可サーバーを使用しています。なぜこの用語を選んだのかと思われるかもしれませんが、この後で説明するとおり、納得のいく理由があります。

たとえば、認証されたユーザーが予約を入れることのできるレストラン向けアプリケーションがあるとします。アプリケーションは、これらのユーザーのクレデンシャルを処理する代わりに、OpenID Connectを使用することで、認証プロセスをアイデンティティプロバイダ（Google、Microsoft、Auth0など）にオフロードします。具体的には、訪問者がこのプロセスを開始すると、アプリケーションは訪問者を選択したアイデンティティプロバイダにリダイレクトさせます。訪問者は、このプロバイダに対して自分自身を認証して身元を証明します。認証プロセスの後、アプリケーションがこの身元証明を取得すると、ユーザーはそれに基づいてレストランに予約を入れられるようになります。

これらのアイデンティティプロバイダが認証プロセスをどのように処理するかは、OpenID Connectのスコープ外です。つまり、プロトコルの観点からみると、多要素認証などの機能を使用してプロセスのセキュリティを強化するかどうか、クレデンシャルのセット（ユーザー名とパスワードなど）を使用してユーザー認証を処理するかどうか、さらには、このプロセスを他のアイデンティティプロバイダや他のプロトコルに中継するかどうかは、アイデンティティプロバイダが自由に選択できます。OIDCが定義するのは、アイデンティティプロバイダとアプリケーションが対話してエンドユーザー認証を安全な方法で確立する方法です。

アプリケーションプロトコルの概説

OpenID Connectに関する入門セクションを読んだ後、「プロトコルが重要なのはなぜか？」、「ユーザー名とパスワードをローカルで処理し続けるだけではなぜいけないのか？」、「これとまったく同じような問題を解決する他のプロトコルがあるのに、それらを使用しないのはなぜか？」などの疑問が生じたのではないでしょうか。これらの疑問を解消するには、デジタルアイデンティティのシナリオが、長年にわたってどのように進化してきたかについて少し学ぶ必要があります。

最初は混乱状態でした（つまり、パスワードのことです）。人類がコンピュータを発明する前、人々はパスワードを使用して、特定の領域に誰がアクセスできるかを制御していました。最初のコンピュータが開発されると、間もなくしてこれらのコンピュータを保護する方法が必要であることに気付き、「コンピュータでパスワードをサポートすればよいのでは？」と考えました。この新時代の初期の頃は、パスワードを使用することでうまくいっていました。マシンまたはアプリケーションを使用したい人は、パスワードを知っている必要があり、また、物理的にそこにいなければなりませんでした。

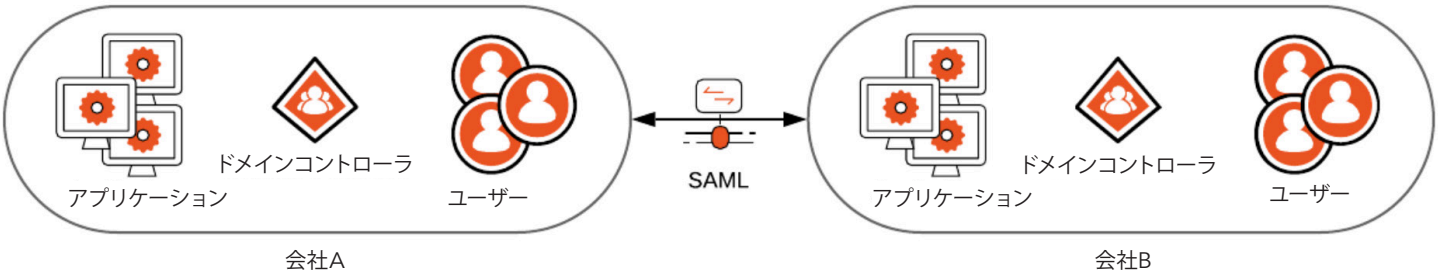
ところがその後、コンピュータネットワークが開発されると、ユーザーが「そこにいる」必要はなくなりました。この変化が起きたとき、世界中の組織はすぐに、コンピュータとアプリケーションに認証を処理させることは、さまざまな理由で（たとえば、ユーザーが至る所で複製されるなどの理由から）あまりよい考えではないことに気づきました。ここで登場したのが、Kerberosなどの最初の認証プロトコルです。

当時、コンピュータサイエンティストは、ドメインコントローラ（ログインやアクセス許可チェックなどのセキュリティ認証リクエストに応答するサーバー）で認証を一元管理することにしました。これらのドメインコントローラの最大の問題（最初のプロトコルで定義）は、プロセスに関与するすべてのエンティティ（エンドユーザー、アプリケーション、サービス）を単一の組織の制御下に置くと考えられていたことです。つまり、これらのプロトコルは、一般的なクラウドコンピューティングのシナリオのように、ユーザーがサードパーティ製クライアントに接続することを求めるシナリオに対応するよう開発されたものではなく、アイデンティティプロバイダを処理する組織には、通常、サードパーティ製クライアントの知識も制御権もありません。

会社A

この計画に不備があり、より優れたソリューションが必要であることにITコミュニティが気付くまでに、そう長くはかかりませんでした。そこで大手ITベンダー数社が、この問題を解決するための委員会を編成しました。この委員会が考案したソリューションは、異なるセキュリティドメイン間で認証および認可データを交換するためのSAML（Security Assertion Markup Language）規格です。SAMLを使用すると、企業Aに所属しているユーザーは、ユーザープロファイルを複製することを要求せずに、企業Bのサービスを信頼性の高い方法で利用できます。このプロトコルは、サービスプロバイダ（この場合はB社）がユーザーアイデンティティをその場で検証することを可能にするよう考案されました。

* **注：**SAMLとOpenID Connectはいずれも、デジタル署名されたトークンを利用して、エンドユーザーの個人属性を保持します。これらのデジタル署名により、サードパーティ製アプリケーションは、情報の真偽をその場で確認できます（つまり、データを確認するための別のリクエストをアイデンティティプロバイダに送る必要はありません）。



SAMLは、OpenID Connectが今日解決している問題の大半を解決しました。ただし、このプロトコルに固有の大きな問題として、XMLに基づく度合いが高いという点がありました。XMLの何が問題かというと、アイデンティティにデジタル署名する必要がある場合に、異なる順序でリストされた2つの要素が署名検証を破る可能性があるという柔軟性がネックになることです。また、XML形式は非常に冗長でもあります。このために、高速接続が利用できない状況や、関連するデバイスがそれほど頑丈でない状況であっても、アプリケーションがただ単にエンドユーザーを認証し、そのアイデンティティを簡単に確認したいだけの場合は、機能が強力すぎるため、処理の負荷がかかりすぎることになります。

そのため、SAMLは技術的にはOpenID Connectと同様のシナリオに対処するのに十分ですが、ほとんどの場合ビジネスの世界で使用されるに至っています。消費者の世界（ソーシャルネットワークなど）では、OAuth（Open Authorization）と呼ばれる別の取り組みが進められました。

OAuthの詳細とそれが重要な理由については、後ほど詳しく説明しますが、要約すると、このプロトコルは、**委任認可**のシナリオをサポートするよう特別に設計されたものです（委任認可とは、たとえば、あたかも自分で行ったかのようにFacebook上で何かを投稿する許可を無作為のアプリケーションに与えることです）。ところが、このプロトコルが広く普及するに伴い、開発者は、エンドユーザーの認証処理など、本来意図したものとは異なる目的にプロトコルを使用し始めました。これを受けて生まれたのが、認証に対応できるようOAuth 2.0を拡張したプロトコルのOpenID Connectです。

上記のストーリーは、かなり短くまとめられていますが、このセクションの冒頭にあった次の疑問を解消するのに十分な情報を含んでいます。

- ✓ **認証プロトコルが重要なのはなぜか？**主な理由、それは開発者はアイデンティティ管理の問題を安全で相互運用可能な方法で解決したいからです。
- ✓ **ユーザー名とパスワードをローカルで処理し続けるだけではなぜいけないのか？**ほとんどの場合、ユーザーはアプリケーションにアクセスするために既存のアカウントを再利用する必要があり、これらのアプリケーションは他の場所のユーザーに対してアクションを実行する必要があるためです。
- ✓ **これとまったく同じ問題を解決する他のプロトコルがあるのに、それらを使用しないのはなぜか？**OpenID Connectが使用されているシナリオ（クラウド環境など）を技術的にサポートしていないか、問題を解決するためのコストがかかりすぎる可能性があるためです。

OpenID Connectの仕組み

OpenID Connectプロトコルがどのように機能するかの分析は、エンドユーザーの視点とソフトウェアの視点の2つの異なる視点から行います。エンドユーザーの視点から見ると、OIDCフローに含まれる手順は非常にシンプルです。まず、ユーザーがアプリケーションを開いて認証プロセスを開始します。ユーザーがこのプロセスを開始すると、アプリケーションがユーザーをアイデンティティプロバイダにリダイレクトさせ、ユーザーはサービスによってリクエストされた手段で認証を行います。認証後、アイデンティティプロバイダはユーザーをアプリケーションに再びリダイレクトさせ、ユーザーがアプリケーションにログインされます。

上記の説明はプロセスを過度に単純化していますが、エンドユーザーの視点から何が見えるかを説明するには十分です。ただし、この後で学習するとおり、OpenID Connectを使用してユーザーを認証するアプリケーションの開発には、より多くの手順と不確定要素が伴います。また、これらの手順はアプリケーションが実行されるプラットフォームによって異なります。

たとえば、従来のウェブアプリケーション（ページがリクエストされるたびにブラウザで完全な再読み込みを行うアプリケーション）を考えてみましょう。このタイプのアプリケーションの視点から見た場合、ユーザーの認証に必要な手順は次のようになります。

1. 訪問者が認証プロセスを開始するようアプリケーションにリクエストする
2. アプリケーションが訪問者をアイデンティティプロバイダにリダイレクトさせる
3. アイデンティティプロバイダが少数のアイテムとともに訪問者をアプリケーションに再びリダイレクトさせる
4. アプリケーションがこれらのアイテムを使用して、認証プロセスを完了するようアイデンティティプロバイダにリクエストを送る
5. アプリケーションが、ユーザーがログイン済みであることを示す何らかの情報を含むページを ユーザーに表示する

* **重要な用語：**認証プロセスの終了時にアプリケーションが取得する重要なアイテムの1つは、IDトークンです。このトークンには、ユーザーに関する個人属性のセットが含まれるため、アプリケーションはそれを使用してユーザー（すなわち、名前とIDトークン）を識別することができます。

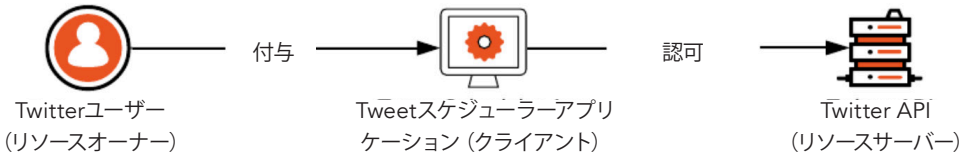
プロトコルを深く掘り下げると、上記の手順も過度に単純化されていることがわかります。ただし、認証プロセスがどのように流れ、アプリケーション開発者の目にどのように映るかは、この手順からおおむねわかります。

簡単に言及したことの1つとして、OpenID Connectが委任認可に対応したOAuth 2.0と呼ばれるフレームワークに基づいていることがあります。このフレームワークがどのように機能するかを知っている人であれば、上記の手順がOAuth 2.0の認可フローで行われることとよく似ていることにおそらく気づいたでしょう。プロセス間の類似性は偶然ではなく、一方が他方を拡張するという事実を反映しています。次のセクションでは、OAuth 2.0を取り上げ、このフレームワークがどのように認可を処理するかについて簡単に説明します。

OAuth 2.0の概要

公式の用語を使用すると、OAuth 2.0とは、**クライアント**が**リソース所有者**に代わって**リソース**サーバーを使用することを可能にする認可フレームワークです。クライアント、リソースサーバー、リソース所有者という用語は、おそらくなじみ深いものではないため、このフレームワークの機能を説明する助けにはならないかもしれません。

その場合のために、次の例を考えてみることにしましょう。あるユーザーが「ツイート」ボタンを押すために「そこにいる」必要がないよう、ツイートをスケジュールできるサードパーティ製アプリケーションを使用しているとします。このシナリオでは、ツイートを投稿するアプリケーションがOAuth 2.0でいうところのクライアントとなり、ユーザーがリソース所有者（ツイートの所有者）となり、Twitter APIがリソースサーバー（ツイートが存在する場所）となります。



このシナリオを念頭に置き、フレームワークで使用されている用語を、広く知られている用語に置き換えると、定義は次のようになります。OAuth 2.0とは、アプリケーションがユーザーに代わってAPIを使用することを可能にする認可フレームワークです。この定義は技術的な精密さは決して高くありませんが、セクションの冒頭に示した定義よりは多少理解しやすいのではないでしょうか。

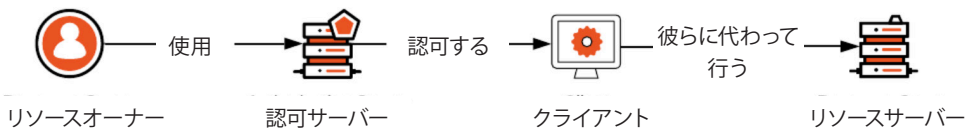
上記の要素に加え、OAuth 2.0フレームワークが定義しているもう1つの重要な用語は、**認可サーバー**です。この章の冒頭で述べたとおり、OpenID Connectでは、ユーザーの認証を担当するエンティティを指す用語として認可サーバーを使用しています。その背景には、エンドユーザーがサインインする場所としての役割を含めるよう認可サーバーという用語がOIDCで転用（または拡大解釈）されるようになったことがあります。ただし、純粋な意味のOAuth 2.0では、このエンティティが果たす役割はわずかに異なり

ます。OAuth 2.0フローの認可サーバーを使用することで、リソース所有者は、自分に代わって何かを行うための権限をクライアントに付与するかどうかを決定できます。

ユーザーに代わってツイートする許可をアプリケーションに与えるシナリオをもう一度取り上げると、Twitter自体が認可サーバーとなります。この場合、ユーザーが初めてツイートをスケジュールするようサードパーティ製アプリケーションに要求したときに、ユーザーはアプリケーションによってTwitterにリダイレクトされ、この同意をアプリケーションに与えるかどうかを尋ねられます。ユーザーが同意した場合、Twitterは許可が与えられたことを示すアイテムをアプリケーションに提供します。次に、スケジュールされたツイートのうちの1つを投稿する時間になると、サードパーティ製アプリケーションはTwitter APIにリクエストを送信し、アイテムを提示して、このアクションを実行する許可をアプリケーションに与えたことをリソースサーバーに通知します。

* **重要な用語：**OAuth 2.0フレームワークでは、ユーザー（リソース所有者）に代わってアクションを実行するための委任認可をサードパーティ製アプリケーション（クライアント）に与えるアイテムを、**アクセストークン**として定義しています。この名前はその意味をよく表しています。API（リソースサーバー）は、その一部にアクセスして何らかのアクションを実行することをアプリケーションに許可するために、このトークンを使用します。

上記のシナリオの最初の部分（自分に代わってツイートすることをサードパーティ製アプリケーションに対して許可することをTwitterに伝える部分）は、OpenID Connectを使用したエンドユーザーの認証についてこれまでに学んだことと似ています。このステップでは、ユーザーが認可サーバーにリダイレクトされ、そこで何らかのアクションを実行すると、アプリケーションはユーザーに関連するアイテムを取得します。最大の違いは、OAuth 2.0の純粋なシナリオでは、結果が個人属性を含んだアイテムではなく、委任認可を与えるアイテムになるということです。



このセクションでは、OAuth 2.0の機能とその仕組みについて要約しています。このフレームワークがどのように機能するかについての詳細をすべて学ぶには、本書以外の書籍も読むことが必要となりますが、重要なポイントは次のとおりです。

- ✓ OAuth 2.0は、委任シナリオに対応する認可フレームワークであり、あらゆる種類の認可シナリオを網羅しているわけではありません。
- ✓ （前の章の演劇のチケットを購入するシナリオで説明したとおり）OAuth 2.0が処理するのは認可のみであるため、エンドユーザーの認証を単独ではサポートしません。

OpenID Connectのユースケース

OpenID Connectに関する知識が増えてきたところで、ここでは、どのようなシナリオでこのプロトコルを利用できるかについて学習します。基本的には、OIDCの使用が適しているケースが3つあります。

まず、OpenID Connectを使用することで、アイデンティティプロバイダで利用しているアカウントをユーザーが再利用可能にするケースです。つまり、もう1つ別のアカウントを作成するようユーザーに要求するのではなく、OIDCを利用してGoogleやMicrosoftなどのアイデンティティプロバイダと連携し、既存のアカウントをユーザーが再利用できるようにすることができます。前者の場合は、ユーザーが記憶しなければならないクレデンシャルがまた1つ増え、さらにひどい場合は、ユーザー名とパスワードが再利用されかねません。

このシナリオには、組織とそのユーザーにとって複数のメリットがあります。まず、前述したとおり、ユーザーはもう1つ別のアカウントを作成する必要がありません。これは、サインアッププロセスがよりスムーズになり、プロセスの中途での離脱が少なくなることも意味します。また、OIDCをこのように使用すると、アプリケーションがユーザーに関する個人情報を要求しやすくなります。この後で学習するとおり、OpenID Connectでは、ユーザーに関する詳細情報を取得するために使用できる一連のプロファイルデータカテゴリが定義されています。

OpenID Connectが役立つもう1つのシナリオは、プロトコルを使用してアイデンティティプロバイダのハブを作成する場合です。このシナリオでは、アプリケーションは複数のプロバイダと通信する代わりに、他のプロバイダのハブとして機能する単一のプロバイダに接続できます。たとえば、Auth0のドキュメントを確認すると、Auth0がこのシナリオをサポートしており、50を超えるアイデンティティプロバイダと連携していることがわかります。ご想像のとおり、ハブとして機能する単一のアイデンティティプロバイダをサポートする方が、それぞれを個別にサポートするよりもはるかに簡単です。

OpenID Connectが非常に役立つ3番目のシナリオは、OpenID Connectが他のプロトコルのプロキシとして機能する場合です。たとえば、OpenID Connectのアイデンティティプロバイダを、SAMLなどのより限定的なプロトコルのプロキシとして機能させることができます。ここでの利点は、このアプローチを用いることで、リソースに制約のあるデバイスをOpenID Connect経由でSAMLアイデンティティプロバイダと連携させられることです。

このように、OpenID Connectは、アイデンティティ管理をより簡単で拡張性の高いものにできるさまざまな可能性をもっています。これにより、プロセスがIT部門にとってより容易で一元的なものになり、アプリケーションを利用したいと考えているユーザーからの要求度が低くなるため、ユーザーディレクトリへの対処に役立ちます。

まとめ

この章では多くの情報を取り扱ったため、次章に進む前に主要なポイントを要約しておきましょう。まず、OpenID Connectを使用すると、アプリケーションが認証プロセスの負担をアイデンティティプロバイダにオフロードできることを学びました。その後、認証プロトコルの進化とOpenID Connectが開発された理由について簡単に確認しました。次に、プロトコルがエンドユーザー認証にどのように対処しているかについて、さらに詳細に見ていきました。最後に、OAuth 2.0についておよびOAuth 2.0がOpenID Connectとどのような関係にあるかについて学びました。

これらのセクションでは、OpenID ConnectとSAMLやOAuth 2.0などの他のプロトコルの両方で定義されている重要な用語も紹介しました。その中でも特に重要なのは次のとおりです。

- ✓ **認可サーバー** – 純粋なOAuth 2.0認可フローでは、エンドユーザーが自分に代わってアクションを実行することをサードパーティ製アプリケーションに認可する場所。または、OpenID Connectフローでは、エンドユーザーが認証する場所。
- ✓ **IDトークン** – OpenID Connectフローで認証するエンドユーザーに関する個人情報を保持するアイテム。
- ✓ **デジタル署名** – サードパーティ製アプリケーションがIDトークンに含まれている情報の真偽を確認できるメカニズム。
- ✓ **委任認可** – エンティティが自分に代わってアクションを実行することを別のエンティティに対して許可する特定の種類の認可。
- ✓ **アクセストークン** – リソース所有者（エンドユーザーなど）に代わってアクションを実行することをサードパーティ製アプリケーションに認可するアイテム。

これらの知識をもとに、実際の使用方法について学んでいきましょう。

OpenID Connectの実行

この章では、さまざまな種類のアプリケーションでOpenID Connectを使用する方法について学習します。ここでの目標は、認証プロセスの仕組みを最初から最後まで確認し、このプロトコルを使用してユーザーをログインさせるのに必要な手順を習得することです。この知識があれば、OpenID Connectをより深く理解し、期待どおりに機能しない場合にデバッグを実行できるようになります。

もう1つの目標は、SDKを使用してプロセスをより簡単に進める方法を示すことです。Auth0やGoogleなどのアイデンティティプロバイダを使用すると、ここに示すほとんどの手順を、サポートされるSDKにオフロードできます。実際、プロトコルのすべての詳細をコーディングして処理する代わりに、これらのSDKを使用することが推奨されています。そうすることで、開発の負担が軽くなるだけでなく、これらのSDKはアイデンティティの専門家によって常にチェックされているため、アプリケーションのセキュリティも強化されます。

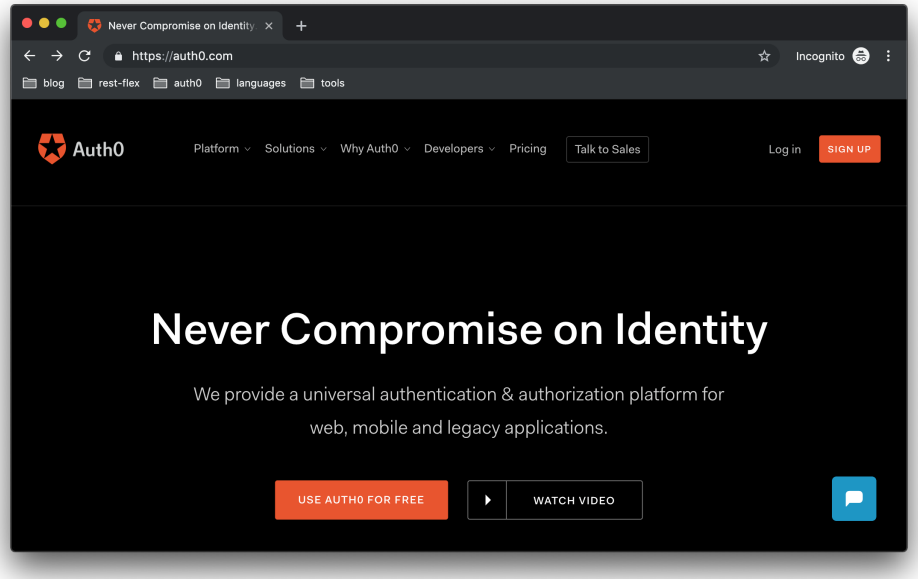
このことを念頭に置いて、この章は次のように進めていきます。まず、無料のAuth0アカウントを作成して、OpenID Connectプロバイダとして使用できるようにします。次に、この章のさまざまなセクションで必要になる依存パッケージをインストールします。そして、通常のウェブアプリケーションでOpenID Connectを使用して認証を処理する方法について学びます（シングルページアプリケーションとネイティブアプリケーションについては、近日中に公開予定の本書の次のバージョンで説明します）。今後参照しやすいよう、各アプリケーションタイプは独自のセクションで説明されています。

* **注：**OpenID Connectは複数のアイデンティティプロバイダが準拠している規格ですが、ここに示すコードをAuth0以外のプロバイダで使用しようとすると、実装に多少の違いが生じ、それが問題になる可能性があります。ただし、試していただくことはできます。試された方がいらっしゃいましたらぜひ結果をお聞かせください。

Auth0アカウントの作成

さまざまなタイプのアプリケーションがOpenID Connectを処理する方法に集中できるようにするためには、このプロトコルをサポートするアイデンティティプロバイダが必要です。このタスクに使用できるプロバイダは少なくともいくつかあります（GoogleやMicrosoftなど）。その中でも特に広く使用されているのが、豊富なコミュニティ、詳細なドキュメント、十分な無料利用枠を備えたサービスとしてのアイデンティティプロバイダであるAuth0です。

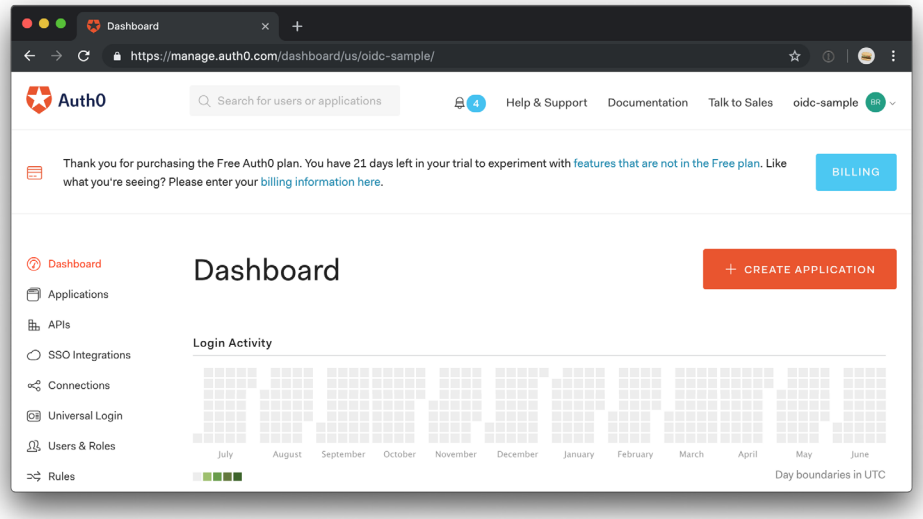
Auth0アカウントをまだお持ちでない場合は、当社のウェブサイトにアクセスして「サインアップ」ボタンをクリックしてください。



このときに、アカウントの作成に使用できるいずれかのオプションを選択します（たとえば、既存のGoogleプロファイルまたはMicrosoftプロファイルを再利用できます）。次の画面で、アイデンティティプロバイダのドメイン（たとえば、oidc-sample.auth0.com）とテナントの地域（本書の執筆時点で、選択できるオプションは欧州、オーストラリア、米国）を選択します。これらのオプションを選択した後で、「次へ」ボタンをクリックして次のステップに進みます。

* **注：**どの地域を選択すべきかわからない場合は、地理的に自分とサーバーの近くにある地域を選択してください。こうすることにより、アプリケーションとアイデンティティプロバイダとの間の待ち時間が短くなり、プロセス全体の速度が向上します。

次のページでは、アカウントタイプを定義するようにAuth0から求められます。希望するオプションを自由に選択し、必要な情報を入力して、「アカウントの作成」をクリックします。このボタンをクリックすると、アカウントのダッシュボードにリダイレクトされます。ダッシュボードにリダイレクトされたということは、この後のセクションで使用するOpenID Connectプロバイダがあることを意味します。



前提条件

以降のセクションでは、OpenID Connectの内部の仕組みを理解できるよう、コードについて掘り下げて説明します。セクションの流れをできる限りスムーズにするため、今日利用可能なテクノロジのうち最も広く普及しているものと言っても過言ではない、Node.jsとNPMを使用します。つまり、JavaScriptを使用して、さまざまなタイプのアプリケーションにOpenID Connectを実装します。また、新しいアプリケーションのスキャフォールディングなどの平凡なタスクに時間を費やさなくて済むように、GitHubを使用して既存のコードを取得します。

Node.jsとNPMは、この後に実行するタスクに不可欠です。したがって、実際に試してみたい場合は、ご使用の環境にインストールする必要があります。これらのツールが使用可能かどうかわからない場合は、ターミナルを開いて次のコマンドを発行します。

```
node -v
npm -v
```

「コマンドが見つかりません」またはそれに類似したメッセージが返された場合は、Node.jsおよびNPMのドキュメントに進み、手順に従ってこれらのツールをインストールします。それに加えて、GitHubから既存のコードを取得するには、2つのオプションがあります。1つは、セクションが表示されるようにGitコマンドラインインターフェイスを使用することで、もう1つは、GitHubのウェブサイトを使用することです。いずれの選択肢も有効であり、行うべきタスクを実行します。ただし、ウェブページを使用するよりも、Git を使用する（ご使用の環境で使用可能にする必要がある）方が簡単でわかりやすいでしょう。

OpenID Connectと従来のウェブアプリケーション

このセクションでは、OpenID Connectを使用して従来のウェブアプリケーションでエンドユーザー認証を処理する方法を学習します。まず、アプリケーションをOpenID Connectプロバイダと統合する複雑な方法を説明した後で、次に、公式SDKを使用することで、これをどのように簡単にできるかを示します。

このセクションで使用するウェブアプリケーションには、ユーザーがOpenID Connectプロバイダを介して認証し、認証プロセスの結果を確認するために使用できるエンドポイントがいくつか含まれています。つまり、認証後、ユーザーはプロバイダによって生成されたIDトークンとこのトークンのコンテンツにアクセスできます。

新しいアプリケーションのスキャフォールディングなどの平凡なタスクに時間を費やさなくて済むように、この章に必要な基盤（依存パッケージや基本ファイルなど）を持つアプリケーションを取得します。次に、この基盤の上に、認証プロセスを最初から最後まで実行するために必要なコードをすべて追加します。

要約すると、このシナリオは以下の手順で構成されます。

1. 基本ファイルと依存パッケージのリストを含むGitHubリポジトリをダウンロードする
2. 依存パッケージをローカルにインストールする
3. Discovery EndpointからOIDCプロバイダに関する情報を取得する
4. 認証プロセスをトリガーする/loginエンドポイントを実装する
5. 認証の/callbackを処理する（つまり、IDトークンを取得する）
6. IDトークンを検証して、ユーザーが/profileエンドポイントを通じてトークンのコンテンツを 表示できるようにする

OpenID Connectと認証フロー

先に進む前に、まず知っておくべきことの1つは、さまざまなユーザー認証方法がOpenID Connectプロトコルによって定義されていることです。このセクションでは、それらのうち2つについて学びます。1つはインプリシットフロー（重要な追加機能であるフォーム送信とともに使用します。その理由については、シングルページアプリケーションに関する章で学習します）で、もう1つは認可コードフローです。これらのフロー間の主な違いは、アプリケーションがIDトークンとアクセストークンを取得する方法です。

インプリシットフローでは、トークンは（認可サーバーがユーザーをアプリケーションにリダイレクトさせるときに）認可エンドポイントによってアプリケーションに直接渡されます。認可コードフローでは、アプリケーションは最初に認可コード（その名前）を取得した後で、これらのコードを必要なトークンと交換する必要があります。

プロトコル統合の複雑な方法（つまり、SDKを使用しない方法）について学習するときは、インプリシットフローを使用します。このフローを使用すると、他のフローで必要となる余分なステップ（アプリケーションが認可コードとトークンを交換するステップ）を1つスキップできるため、プロセスが少し簡単になります。

Auth0でサポートされている公式SDKについて学習するときは、認可コードフローを使用します。このフローには余分な手順が1つ必要ですが、アプリケーションをOIDCプロバイダと統合するために必要な労力は、手動で行う場合と比べて最小限に抑えられていることがわかります。

プロジェクトのブートストラップ

GitHubからダウンロードするリポジトリは、Node.jsとExpressを使用してウェブサーバーを定義します。Node.jsとNPMはすでにマシンにインストールされているので、先に進んでリポジトリを取得することができます。Gitがローカルにインストールされている場合は、プロジェクトを保存するターミナルを開き、次のコマンドを発行します。

```
git clone https://github.com/auth0-blog/oidc-book-regular-
webapp. git
cd oidc-book-regular-webapp
```

Gitがインストールされていない場合は、このURLに移動し、緑色のボタンを使用してプロジェクトをダウンロードします。プロジェクトをダウンロードしたら解凍して、プロジェクトのルートパスを指すターミナルを開きます。

次に、プロジェクトルート内で、NPMを使用して依存パッケージをインストールします。

```
npm install
```

すべてが適切にインストールされていることを確認するには、npm startを実行し、ブラウザでhttp://localhost:3000を開きます。期待どおりに機能している場合は、このプロジェクトの内容を説明する画面と、エンドユーザーに認証プロセスをトリガーさせるために使用する壊れたリンクが表示されます。

通常のウェブアプリケーションをAuth0に登録する

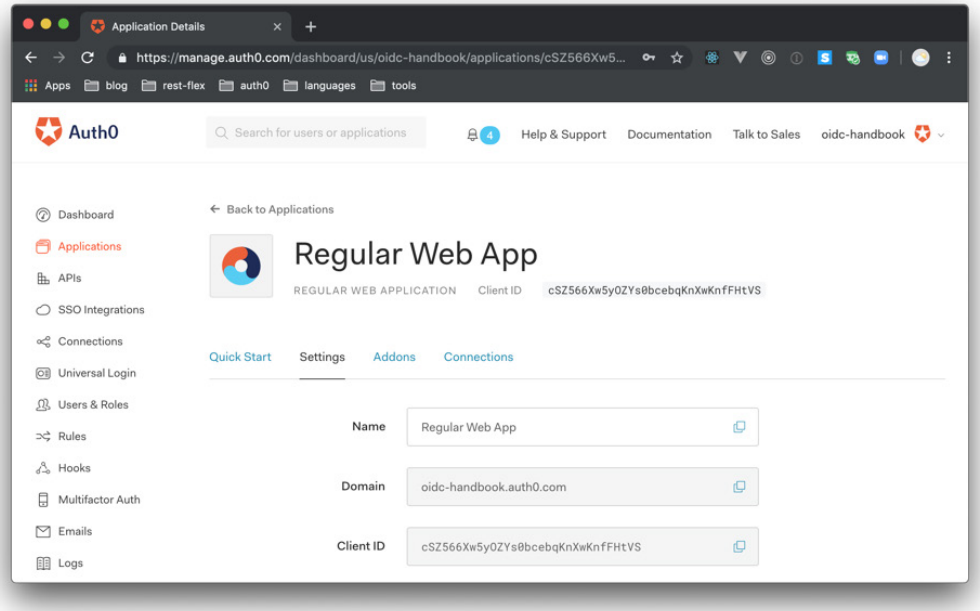
ウェブアプリケーションがOpenID Connectを使用してエンドユーザーのサインインを処理できるようにするには、まずそのアプリケーションをAuth0に登録する必要があります。このセクションでは、Auth0のダッシュボードを使用して、通常のウェブアプリケーションを登録する方法を示します。

まず、ダッシュボードに移動して、「アプリケーション」セクションをクリックします。セクションが表示されたら、「アプリケーションの作成」ボタンをクリックして、次のようにフォームに入力します。

- ✓ **Name:** OIDC通常のウェブアプリケーション
- ✓ **Application Type**：通常のウェブアプリケーション

次に、「作成」ボタンをクリックし、Auth0が新しいアプリケーションの作成を完了したら、「設定」セクションに進みます。そこで行う必要がある変更は、「許可されたコールバックURL」フィールドにhttp://localhost:3000/callbackを設定することだけです。この設定は、OpenID Connectプロバイダが認証プロセスの成功後に呼び出すことを許可されるURLを制限するため、セキュリティの観点から重要です。

この設定を変更した後、ページの一番下までスクロールし、「変更を保存」ボタンをクリックします。後で情報をコピーする必要があるため、ページを開いたままにしておいて構いません。



Discovery Endpointからの情報取得

プロジェクトが正常に機能し、Auth0に登録されたことを確認したら、IDEまたはテキストエディタ（VS CodeやWebStormなど）でソースコードを開き、OpenID Connectプロバイダとのアプリケーションの統合を開始できるようにします。まず、アプリケーションからDiscovery Endpointにリクエストを発行して、プロバイダから情報（認可サーバーが配置されている場所など）を取得する必要があります。

これを行うには、`src/server.js`ファイルを開き、`app.listen`を含んでいる行を検索します（ファイルの一番下にあります）。OpenID Connectプロバイダに関する情報を取得するコード内にapp.listenの呼び出しをネストする必要があります。この情報はユーザーによる認証を可能にするためにアプリケーションで必要となるため、これらのデータを取得するまでは、サーバーによってユーザーのリクエストがリッスンされるのを防ぐという考え方です。

```
const {OIDC_PROVIDER} = process.env;
const discEnd = `https://${OIDC_PROVIDER}/.well-known/openid-
configuration`;
request(discEnd).then((res) => {
 oidcProviderInfo = JSON.parse(res);
 app.listen(3000, () => {
   console.log(`Server running on http://localhost:3000`);
 });
}).catch((error) => {
 console.error(error);
 console.error(`Unable to get OIDC endpoints for ${OIDC_PROVIDER}`);
 process.exit(1);
});
```

ご覧のとおり、app.listenをカプセル化するコードは、OpenID Connectプロバイダの配下のapp.listenというパスにHTTP GETリクエストを発行します。プロバイダがこのリクエストを正しく処理すると、アプリケーションは必要な情報を含んだストリングレスポンスを取得します。次にアプリケーションは、この情報

を読み取れるようにするために、レスポンスを解析してJavaScriptオブジェクトにし、`oidcProviderInfo`という変数に保存します。一方、アプリケーションがプロバイダデータを取得できない場合、アプリケーションはエラーをログに記録し、失敗コード（`process.exit(1)`）で終了します。

お気づきのとおり、上記のコードでは`OIDC_PROVIDER`と呼ばれる環境変数が使用されています。この変数を設定するには、プロジェクトルート内に.envというファイルを作成し、変数を追加します。次に、プロジェクトルート内にCLIENT_IDという新しい環境変数を追加します。最終的に、ファイルは次のようになります。

```
OIDC_PROVIDER=
CLIENT_ID=
```

アプリケーションをOIDCプロバイダに登録するために必要な環境変数は、この2つだけです。最初のOIDC_PROVIDERは、OpenID Connectプロバイダドメインを指します。2番目のCLIENT_IDには、通常のウェブアプリケーションの登録時にAuth0から提供されるクライアントID値が格納されます。

いずれの値も、Auth0ダッシュボードで開いたままにしたページで確認できます。このページの「Domain」と呼ばれるフィールドの値は、コピーしてOIDC_PROVIDERに追加できます。「ClientID」と呼ばれるフィールドには、`CLIENT_ID`に追加する値が含まれています。

アプリケーションがプロバイダから受け取るデータは、ウェブブラウザで`https://${OIDC_PROVIDER}/.well-known/openid-configuration`を開いて確認できます（`${OIDC_PROVIDER}`を適切な値で置き換える必要があります）。これを行うと、以下の情報が表示されます。

`authorization_endpoint`: エンドユーザーが認証に使用するURL

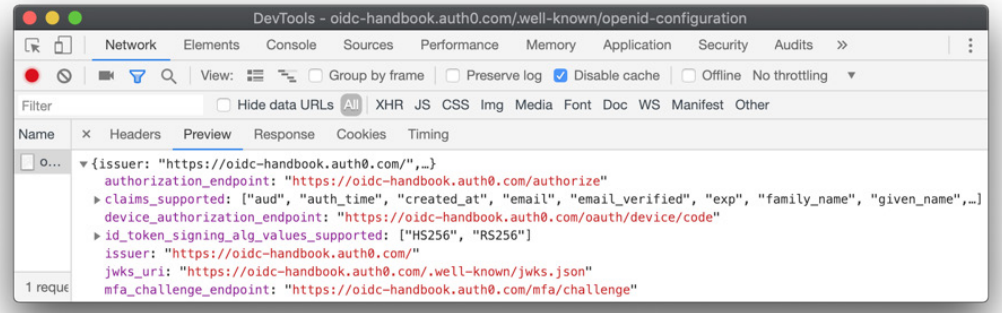
`claims_supported`: サポートされるクレームが含まれているアレイ（クレームについては、後で詳しく説明します）

`issuer`: OIDCプロバイダの識別子（通常はプロバイダのドメイン）

jwks_uri: トークンの検証に使用されるパブリックキーがプロバイダによって公開されている場所

token_endpoint: アプリケーションがトークンを取得するために使用できるURL

userinfo_endpoint: アプリケーションが特定のユーザーの詳細を確認するために使用するURL



認証プロセスの開始

アプリケーションがOpenID Connectプロバイダに登録されたので、次に、`/login`エンドポイントを実装します。これを行うと、ユーザーが認証プロセスを開始できるようになりますが、アプリケーションはまだ認証プロセスを完了できません。これについては、次のセクションで`/callback`エンドポイントを実装するときに説明します。

`src/server.js`ファイルに戻り、`/login`エンドポイント定義を検索します。この時点では、このエンドポイントによってHTTPステータス501（未実装）が返されます。この定義全体を次のコードで置き換えます。

```
app.get(,/login‘, (req, res) => {
  // define constants for the authorization request
  const authorizationEndpoint = oidcProviderInfo[,authorization_
```

```
endpoint‘];
  const responseType = ‚id_token‘;
  const scope = ‚openid‘;
  const clientID = process.env.CLIENT_ID;
  const redirectUri = ‚http://localhost:3000/callback‘;
  const responseMode = ‚form_post‘;
  const nonce = crypto.randomBytes(16).toString(‚hex‘);
  // define a signed cookie containing the nonce value
  const options = {
    maxAge: 1000 * 60 * 15,
    httpOnly: true, // The cookie only accessible by the web server
    signed: true // Indicates if the cookie should be signed
  };
  // add cookie to the response and issue a 302 redirecting user
  res
    .cookie(nonceCookie, nonce, options)
    .redirect(
      authorizationEndpoint +
      ‚?response_mode=‘ + responseMode +
      ‚&response_type=‘ + responseType +
      ‚&scope=‘ + scope +
      ‚&client_id=‘ + clientID +
      ‚&redirect_uri=‘+ redirectUri +
      ‚&nonce=‘+ nonce
    );
});
```

新しい定義の最上部では、認可リクエストを作成するために必要となる以下の定数が初期化されています。

authorizationEndpoint: ユーザーのリダイレクト先にする認可URL

responseType: アプリケーションがプロバイダに期待するレスポンスタイプ

scope: ユーザーの認証について確認する必要のある情報

clientID: プロバイダがアプリケーションに割り当てる識別子

redirectUri: 認証プロセスの後でプロバイダがユーザーをリダイレクトさせる場所

responseMode: エンドユーザーのIDトークンがアプリケーションによって取得される方法

nonce: アプリケーションが反射攻撃を防ぐのに役立つランダムな文字列

* **注**：上記の説明では、「認証リクエスト」ではなく「認可リクエスト」を使用しています。これは、ユーザーが認証に使用するエンドポイントがOpenID Connectで認可サーバーと呼ばれるのと同じ理由によるものです（つまり、このプロトコルはOAuth 2.0の拡張であるため）。

上記のリストには多くの情報が詰まっています。まず、responseTypeをid_tokenに設定します。なぜならば、ここで行う必要があるのは、ユーザーが正常に認証されたことの確認を得ることだけであり、この情報を認証プロセスから直接取得したい（つまり、インプリシットフローを実装したい）からです。このパラメーターにはいくつかの代替オプションがあります。たとえば、id_tokenを要求する代わりに、codeを使用できます。そうすることで、認可コードフローを実装することになり、アプリケーションはプロバイダからcodeのみを取得するようになります。

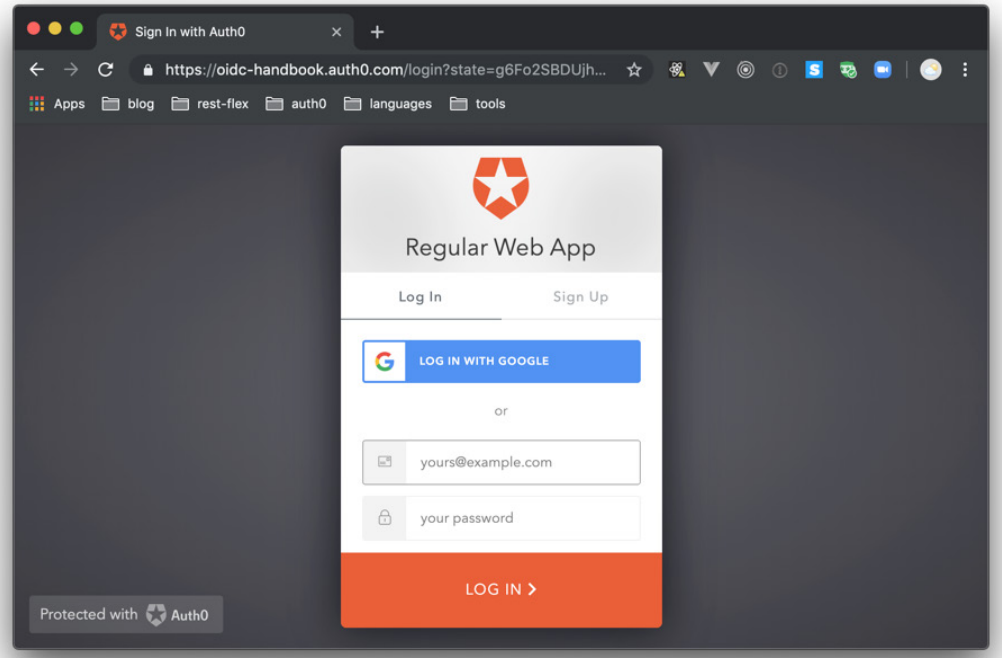
ログインしようとしているユーザーについて個人情報から何を知ることができるかということになると、重要な定数はscopeです。このパラメーターではopenidを使用しているだけなので、ユーザーがログインしたという情報と、プロバイダでの識別子を含むsubクレームのみが取得されます。ユーザーに関する詳細情報を取得するには、スコープを追加する必要があります（たとえば、name、family_name、given_nameなどのクレームを取得するには、profileを追加します）。これが実際にどのように表示されるかについては、この後で説明します。

* **注：** OpenID Connectでは、クレームという用語が、ユーザー情報を伝達する属性の意味で使用されています。認可サーバーは、これらのクレームの発行を担当するエンティティです。クレームを取得する方法は、セキュアチャネル（HTTPS）を使用する方法と、IDトークンを使用する方法の2つだけであるため、アプリケーションはクレームに依存できます。トークンの検証とセキュアチャネルについては、後で学習します。

ここで定義するもう1つ別の重要な定数は、form_postを値として持つresponseModeです。この定数は、アプリケーションでレスポンス（この場合はid_token）を取得したいことをOIDCプロバイダに通知するために、HTTP POSTリクエストの本文で使用します。代替手段は、クエリパラメーターとしてレスポンスを取得するか、またはフラグメント（ブラウザに到達するだけで、サーバーに到達することはない）でレスポンスを取得することです。どちらの手段も、サーバーまたはブラウザのいずれかで記録される可能性があるため、安全性が低くなります。

これらの定数を定義した後、エンドポイント定義はoptionsと呼ばれる別の定数を作成します。この定数は、nonce値（反射攻撃を防止する値）の格納方法を設定するために使用されます。この場合、アプリケーションは、HTTPリクエストでのみ使用でき（つまり、クライアント側のスクリプトでは表示されない）、デジタル署名され、15分間のみ有効な値をCookieに追加します。プロバイダが送信するIDトークンの検証中に、/callbackエンドポイントはCookieからnonce値を読み取るため、このトークンに付加されているものと比較できます。

最後に、新しい`/login`エンドポイントは、呼び出し元にHTTP 302（リダイレクト）レスポンスを発行し、上記の定数によって定義されたパラメーターを使用して認可サーバーに移動する必要があることを示します。これが設定された状態で、アプリケーションを再起動すると、ログインリンクをクリックすることで、リダイレクトが適切に機能しているかどうかを確認できるようになります。すべてが期待どおりに機能している場合は、OpenID Connectプロバイダのログインページが表示されます。


Sign In with Auth0
https://oidc-handbook.auth0.com/login?state=g6Fo2SBDUjh...
Apps
blog
rest-flex
auth0
languages
tools
Regular Web App
Log In
Sign Up
LOG IN WITH GOOGLE
or
yours@example.com
your password
LOG IN
Protected with Auth0

認証コールバックへの対処

ユーザーが認証プロセスを開始できるようにした後、次に対処する必要があるのは認証コールバックです。ユーザーが認証のためのログインページの処理を完了すると、OpenID Connectプロバイダは、認可リクエストのredirectUriパラメーターで渡したURLにユーザーをリダイレクトします。form_postをresponseTypeとして使用するようプロバイダに要求したため、認証サーバーはIDトークンを生成し、それをHTMLフォームに埋め込んで、エンドユーザーのブラウザで表示します。プロバイダが表示するページには、表示されるとすぐにHTMLフォームを自動的に送信するスクリプトも含まれます。

上記のプロセス全体はユーザーに対して透過的に行われますが、IDトークンの取得方法に影響します。つまり、サーバーはHTTP POSTリクエストを処理し、application/x-www-form-urlencodedコンテンツタイプを解析できなければなりません。この例のプロジェクトの場合、このタイプのコンテンツを解析するために必要なツールがすでにあります。必要なのは、/callbackエンドポイント定義を検索して、それを次の値で置き換えることだけです。

```
app.post(,/callback‘, async (req, res) => {
  // take nonce from cookie
  const nonce = req.signedCookies[nonceCookie];
  // delete nonce
  delete req.signedCookies[nonceCookie];

  // take ID Token posted by the user
  const {id_token} = req.body;

  // decode token
  const decodedToken = jwt.decode(id_token, {complete: true});

  // get key id
  const kid = decodedToken.header.kid;
```

```
// get public key
const client = jwksClient({
  jwksUri: oidcProviderInfo[,jwks_uri‘],
});

client.getSigningKey(kid, (err, key) => {
  const signingKey = key.publicKey || key.rsaPublicKey;

  // verify signature & decode token
  const verifiedToken = jwt.verify(id_token, signingKey);

  // check audience, nonce, and expiration time
  const {
    nonce: decodedNonce ,
    aud: audience,
    exp: expirationDate,
    iss: issuer
  } = verifiedToken;
  const currentTime = Math.floor(Date.now() / 1000);
  const expectedAudience = process.env.CLIENT_ID;
  if (audience !== expectedAudience ||
      decodedNonce !== nonce ||
      expirationDate < currentTime ||
      issuer !== oidcProviderInfo[,issuer‘]) {
    // send an unauthorized http status
    return res.status(401).send();
  }
```

```
            req.session.decodedIdToken = verifiedToken;
            req.session.idToken = id_token;

            // send the decoded version of the ID Token
            res.redirect(,/profile‘);
        });
    });
```

これらの変更が適切に行われた状態で、ユーザーが（認証が成功した結果として）このエンドポイントを呼び出すと、認証リクエストに対して生成されたnonceの値を格納する定数が新しいバージョンによって作成されます。この定数の目的は、IDトークンの内容と比較することです。それに加えて、エンドポイントはnonceCookieを削除するため、アプリケーションが反射攻撃にさらされません。次に、エンドポイントはOIDCプロバイダから送信されたIDトークンを読み取り、それをデコードしてトークン内の詳細を確認できるようにします。これらの詳細を入手すると、エンドポイントはIDトークンの署名を検証するプロセスを開始します。

* **注：** IDトークンのデジタル署名の検証は、信頼できる認可サーバーによって実際にトークンが作成されたかどうかを確認するためにアプリケーションが実行する必要のある重要な手順です。ただし、このプロセスが不要なシナリオがいくつかあります。たとえば、IDトークンがバックチャネルを介して取得された場合（つまり、ユーザーのデバイスで実行されているエージェントをトークンが通過しない場合）、アプリケーションはこの手順をスキップして、IDトークンの取得に使用されたチャネル（HTTPS）を信頼します。

IDトークンの署名を検証できるようにするには、アプリケーションが認可サーバーからキーを取得する必要があります。アプリケーションが必要とするキーは、認可サーバーがトークンの署名に使用したキーと対になるものです。これらのキーは、パブリックキー（アプリケーションが使用するキー）およびプライベートキー（認可サーバーのみが知っているキー）と呼ばれます。これらのキーはともに、トークンの署名と検証のための非対称アルゴリズムで使用されます。

トークンの署名の検証に必要なパブリックキーを取得するには、アプリケーションがJWKS URIにリクエストを発行する必要があります。このURIは、これらのキーを含んだキーのセットを返します。このコードでは、URIのコンテンツを解析するプロセスを支援し、適切なキーを見つけるために、jwksClientというオブジェクトを使用しています。このオブジェクトを作成するために渡す必要があるのは、OpenID ConnectのDiscovery Endpoint（oidcProviderInfo[,jwks_uri']）から取得するJWKSのURIそのものと、IDトークンのヘッダーにあるkidプロパティです。

これらの情報がそろったら、次に行うことは、getSigningKeyを呼び出してパブリックキーを取得し、それをjsonwebtokenモジュールのverifyメソッドに渡すことです。IDトークンが対となるキー（つまり、プライベートキー）でデジタル署名されている場合、verifyメソッドはエラーをスローすることなく実行を終了し、デコードされたトークンを送り返します。verifiedToken定数の内容は、decodedTokenと同じです。2つの違いは、前者の場合は有効な署名があるという事実に依拠できるのに対し、後者にはこの保証がないことです（後者が必要となる唯一の理由は、kidプロパティを取得してパブリックキーを見つけられるようにすることです）。

IDトークンの署名を検証するプロセスがすべて完了した後、アプリケーションが次に行わなければならないのは、これらのトークンの4つの重要な側面をチェックすることです。

`audience:` トークンがこのアプリケーション専用に作成されていることを確認することが重要です。したがって、expectedAudience（audクレーム）は、アプリケーションを登録したときにプロバイダによって定義されたCLIENT_IDでなければなりません。

`nonce:` 前述したとおり、反射攻撃を防ぐには、認可リクエストの作成時にアプリケーションが生成したのと同じnonce値が、OpenID Connectプロバイダによって IDトークンに付加されていることを確認することも重要です。

`expiration date:` IDトークンのもう1つの重要な特徴は、現在の時刻がexpクレームで表される時刻より前でなければならないことです。

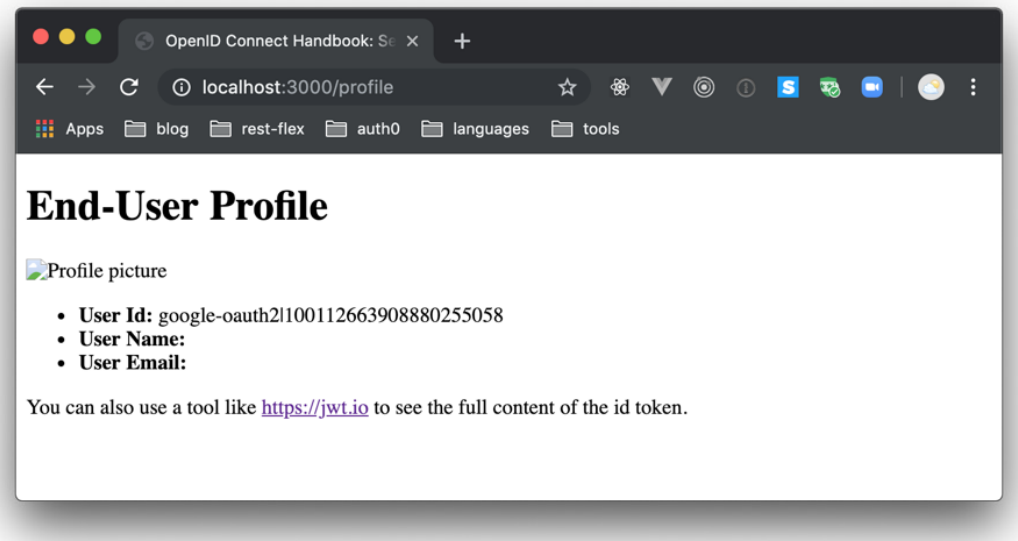
`issuer:` 最後になりますが同様に重要なのは、トークンの発行者（issクレーム）がOIDCプロバイダでなければならないということです。

上記のチェックのいずれかが無効になると、/callbackエンドポイントはIDトークンを拒否し、401 HTTPステータスをエンドユーザーに返します。それ以外の場合、エンドポイントは、decodedIdTokenと元の（エンコードされた）id_tokenという2つの情報をエンドユーザーセッションに追加します。

この情報をセッションに追加することは必要ではなく、OpenID Connectによって指定されるものでもありません（OpenID Connectにとって重要でもありません）。実際、IDトークンを取得して検証した後、この情報をいつどのように使用するかを決定するのはアプリケーション開発者の責任です。この情報は、後で簡単にアクセスできるよう、ウェブアプリケーションによってセッション上に配置されます。

この情報をセッションに追加すると、アプリケーションはエンドユーザーを/profileというエンドポイントにリダイレクトします。このエンドポイントはすでにイニシャルプロジェクトに存在し、デコードされたIDトークンで利用可能な情報の一部を示しています。それに加えて、/profileエンドポイントにはhttps://jwt.ioへのリンクも含まれています。これは、JWT（この場合はid_token）のコンテンツ全体を表示するツールです。

これが実際にどのように表示されるかを確認するには、アプリケーションを再起動してブラウザで開き、ログインリンクをクリックして、OpenID Connectプロバイダに対して認証します。これを行うと、プロバイダはIDトークンを使用してユーザーをアプリケーションに送り返します。次に、上記のチェックを実行した後、/callbackエンドポイントは/profileページにユーザーをリダイレクトします。ここでは、IDトークンの内容の一部とhttps://jwt.ioツールへのリンクを確認できます。



ユーザーに関する詳細情報の要求

アプリケーションを実行してログインした後、/profileページで大量のデータが欠落していることに気付いたのではないでしょうか。たとえば、プロファイル画像が壊れており、電子メールアドレスがありません。ここでの問題は、アプリケーションが認可リクエストの作成時にこのデータを要求していないことです。アプリケーションが要求したのはopenidスコープのみであるため、プロバイダはユーザーの識別子（subクレーム）のみをIDトークンに追加し、それ以外は何もしません。

ユーザーが誰であるかについての詳細情報を取得したい場合は、2つの選択肢があります。最初の選択肢は、より豊富な情報を含んだIDトークンを取得することです。これについては、このセクションで学習します。2番目の選択肢は、アクセストークンに依存するUserInfoというエンドポイントにリクエストを発行することです。2番目のアプローチについては後で学びます。

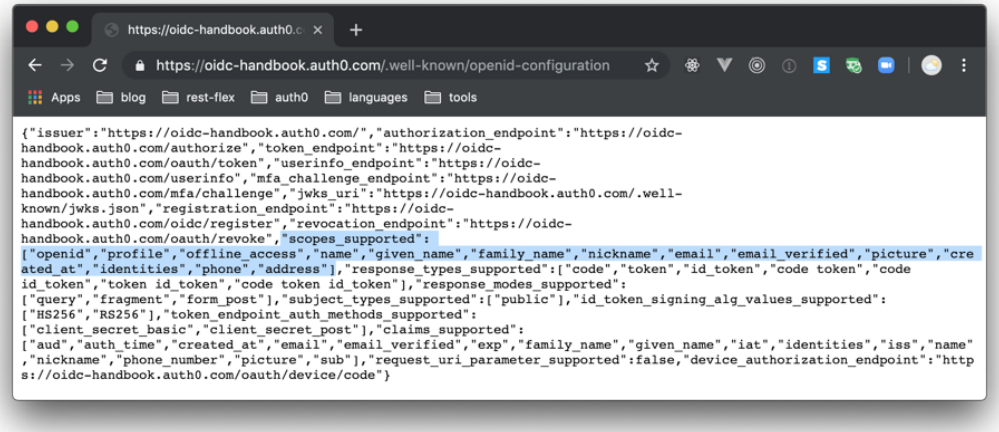
最初のアプローチを実機で確認するには、src/server.jsファイルを開き、scope定数を検索します。この時点では、この定数にはopenid値のみが含まれています。ユーザーに関する詳細情報を取得するには、OpenID Connectプロバイダによってサポートされるスコープの1つを追加する必要があります。プロバイダによってサポートされるスコープを確認するには、Discovery Endpointをもう一度使用します。ウェブブラウザでhttps://${OIDC_PROVIDER}/.well-known/openid-configurationを開くと（${OIDC_PROVIDER}を適切な値で置き換えることを忘れないでください）、scopes_supportedプロパティが表示されます。このプロパティには、少なくともopenid（OIDC仕様で説明されているように必須）と、おそらく次のような他のスコープを含んだ配列があります。

`profile`: name, family_name, and birthdate などのクレームへのアクセスを要求するスコープ

`email`: emailおよびemail_verifiedクレームへのアクセスを要求するスコープ

`address`: addressクレームへのアクセスを要求するスコープ

`phone`: phoneおよびphone_verifiedクレームへのアクセスを要求するスコープ



つまり、サポートされているスコープのいずれかをscope定数に追加しておくと、ユーザーがアプリケーションにサインインしたときに、ユーザーに関する詳細情報を取得できるようになります。たとえば、scope定数のopenidのすぐ右にprofile emailを、スペースで分けて追加します（結果として、この定数の値がopenid profile emailになります）。そして、アプリケーションを再起動して、もう一度認証します。期待したとおりに動作すれば、/profileに移動したときに、リクエストした情報が表示されます。また、https://jwt.ioのリンクをクリックすると、新規IDトークン内のクレームがさらに多く表示されます。

ユーザー認証にSDKを使用する

OIDC認証を、ウェブアプリケーションで扱うために必要な技術的手順をどう考えますか？手順の数は天文学的ではないものの、詳細は決して単純とは言えません。また、慎重に分析すれば、上記のコードが（jsonwebtokenのような）既存のパッケージを利用して、OpenID Connectの動作に関わるたくさんの処理を扱っているかが分かります。たとえば、トークン署名認証について知る必要がある場合も、トークンを確実に署名するためだけに、非常にたくさんの手間が必要になります。つまり、前のセクションはこのプロトコルと関連する技術の導入にすぎないということです。

それを念頭に、このセクションではOpenID ConnectをSDKにより簡単に使用する方法を学びます。ここでは、Node.jsで最も人気がある認証ミドルウェアのpassportと、このプラットフォームのために開発された、標準的なウェブアプリケーションのための公式Auth0ライブラリであるpassport-auth0を使用しま

す。これから見ていくように、主にこのプロトコルで必要となる、エラーを頻発させる技術的詳細に時間を費やす必要がないため、SDKを使用すると作業はより簡単になります。

これらのSDKを利用する方法を学習するために、上記のものと似た手順を追っていきます。ゼロからの設定の手間を省くため、既存のプロジェクトを取得します。次に、いくつかのコードを使用して、アプリケーションをOpenID Connectと統合します。Auth0ダッシュボードにすでにアプリケーションを登録しているため、それを再度行う必要はありません。

そのため、追加の手順なしで、新規ターミナルを開き、プロジェクトを保存したディレクトリに移動して、次のコマンドを実行します（代わりに、GitHubの緑のダウンロードボタンを使用して、プロジェクトをローカルに展開することもできます）。

```
git clone https://github.com/auth0-blog/oidc-book-regular-
webapp- auth0.git
cd oidc-book-regular-webapp-auth0
```

次に、任意のIDEにコードを読み込んだ後、src/server.jsファイルを開き、Express serverを作成している行を検索します（const appを検索）。この行のすぐ下に、次のコードを追加します。

```
// Configure Passport to use Auth0
const auth0Strategy = new Auth0Strategy(
  {
    domain: process.env.OIDC_PROVIDER,
    clientID: process.env.CLIENT_ID,
    clientSecret: process.env.CLIENT_SECRET,
    callbackURL: ,http://localhost:3000/callback‘
  },
  (accessToken, refreshToken, extraParams, profile, done) => {
    profile.idToken = extraParams.id_token;
    return done(null, profile);
```

```
    }
);
passport.use(auth0Strategy);
```

追加したコードは、OIDCプロバイダの詳細を持つauth0Strategyと呼ばれる定数を作成し、この定数で使用するpassportを設定します。アプリケーションのpassportを設定する行が他にもいくつかありますが（ユーザーのシリアル化とそれのExpressでの初期化など）、それほど関連しないため省略します。SDKを設定してアプリケーションをプロバイダに統合するために使用するプロパティと、そこから得られる情報（主にトークン）で何を行うかが重要です。

プロパティと関連して、以前行ったことと今回の間には1つの大きな違いがあります。ここでは、それはCLIENT_SECRETプロパティの導入です。以前に構築したアプリケーションでは、IDトークンを認可エンドポイントから直接取得していました（インプリシットフローを実装しました）。そのため、このプロパティは必要ありませんでした。しかし、今回はpassport-auth0 SDKが認可コードフローを実装するため（アプリケーションが認証コードを取得して、それをトークンと交換）、この値も渡す必要があります。

今回は、passport-auth0から取得する情報について言えば、2つのみ使用していることが分かります。

Discovery Endpointの統合 – passport-auth0 SDKは、Auth0Strategy初期化中に渡されるdomainプロパティを使用してこの情報を取得

responseMode value の値 – SDKがアプリケーションのための値を代わりに定義

nonce の値 – これもSDKがアプリケーションのため代わりに対応

認可リクエスト – リクエストURLを自分で構築する代わりに、SDKが代わりにこれを操作

この手順を完了するため、.envファイルをプロジェクトのルートに作成します。そして、次の環境変数をそこに追加します。

```
OIDC_PROVIDER=
CLIENT_ID=
CLIENT_SECRET=
```

次に、Auth0ダッシュボードに以前作成したアプリケーションの、「Settings」タブを開いて、次を使用します。

OIDC_PROVIDER 変数を設定する**Domainプロパティ**

CLIENT_ID 変数を設定する**Client IDプロパティ**

CLIENT_SECRET 変数を設定する**Client Secretプロパティ**

それを設定して、src/server.jsファイルをもう一度開き、/loginと/callbackのエンドポイント定義を次のように変更します。

```
app.get(
  ,/login‘,
  passport.authenticate(,auth0‘, {
    scope: ,openid email profile‘
  }),
);

app.get(,/callback‘, (req, res, next) => {
  passport.authenticate(,auth0‘, (err, user) => {
    if (err) return next(err);
    if (!user) return res.redirect(,/login‘);

    req.logIn(user, function(err) {
      if (err) return next(err);
```

```
            res.redirect(,/profile‘);
          });
        })(req, res, next);
      });
```

ご覧のように、/loginエンドポイントで必要なのは、ユーザーについて知りたい情報を定義し（scopesプロパティの定義）、処理をSDKに任せること（passport.authenticateの呼び出し）だけです。これにより、ユーザーの/loginに対するリクエスト送信時には、SDKが代わりに認可リクエストを作成し、ユーザーに送信するようになります。

/callbackエンドポイントについては、コードはそれほど複雑ではありません。実際、このエンドポイントのコードはSDKの同じメソッド（passport.authenticate）を呼び出して、認証コールバックを処理しています。このエンドポイントでの違いは、認証処理の後、OIDCプロバイダがユーザーに送信を返す際の動作を定義する、コールバック関数を渡すところにあります。

- ✓ エラーが発生すれば、このコードはエラーをnextハンドラーに渡します。このエラーを扱う方法は、あなた次第です（エラーをログに記録、ユーザーに表示などが可能です）。
- ✓ エラーが発生しなかった場合、コードはreq.logInを呼び出し（logInメソッドはpassport SDKの一部です）、現在のセッションにユーザーを設定し、ユーザーを/profileにリダイレクトします。

これらの変更を適用した後、npm startをターミナルで実行して、ログインのためのアイデンティティプロバイダを使用すれば、前のセクションの最後で取得したものと同じページがまた取得されることが分かります。今回の違いは、コードがより少なくなっていることです。また、公式のSDKに頼ることで、継続的にメンテナンスされ、かつメンテナンスで改善された点を活用するためにコードを簡単にアップグレードできるので安心です。

まとめ

このセクションでは理解しておく必要がある多数のコンセプトを紹介しました。始めたばかりの人は、OpenID Connectプロトコルは、エンドユーザー認証を扱うために利用できる、いくつかの異なったフローを定義することを学習しました。具体的には、認可エンドポイントから直接トークンを返すインプリシットフローについて学習し、認可コードを代わりにアプリケーションに渡す認可コードフローについて学習しました。この最後のフローの内部詳細について見る機会はありませんでしたが（SDKが代わりに処理しているため）、後のユーザーの代わりにAPIを利用する方法を学ぶ際に、それが作動しているところを見ていきます。

異なったフローについての学習の後、このセクションではDiscovery Endpointを紹介しました。特に、エンドポイントがどこにあるかと、どのようにエンドポイントがOpenID Connectプロバイダの重要な特性を公開しているかについて学びました。

そして、認可リクエストの構築方法（エンドユーザーが認証するURLを参照）と、認証コールバックの処理方法を学びました。これらのトピックの学習を通じて、トークン認証、リダイレクトURI、ノンス、スコープ、レスポンスタイプ、およびその他の重要なコンセプトに触れました。

最後に、SDKを活用してOpenID Connectプロバイダとの統合をより簡単に行う方法について見ていきました。この知識を身に付けたことで、トピックの入門段階を終えたと見なせるため、より高度なトピックとシナリオに移る準備ができたことになります。

従来のウェブアプリケーションと委任認可フロー

これまでの章で、OpenID Connectを使用して次の2つのことをウェブアプリケーションに実装する方法について学びました。ユーザーのサインインのため認可リクエストを作成するエンドポイントと、認可コールバックを処理してプロファイルを取得し、認可リクエストを完了するエンドポイントの2つです。この知識を身に付けたことで、OpenID Connectの世界の入門段階を終えたと見なせるため、他の関連するトピックについて学ぶ準備はできています。たとえば、もう少し掘り下げる必要がある重要なテーマの1つが、委任認可のテーマです。

「OpenID Connectの導入」の章で、このトピックに少しだけ触れました。おさらいすると、OIDCはOAuth 2.0フレームワークに基づいており、このフレームワークはもともと委任認可シナリオを処理するために開発されたものであると学習しました。また、ユーザーの代わりに情報を利用するには、アプリケーションにはアクセストークンが必要であることも学習しました。この章では、これらのトピックをもう少し深く掘り下げ、アプリケーションがユーザーの情報を利用できるようにするために必要なことを学習していきます。

次のリストはこの章の残りの部分の流れを示しています。

- ✓ 最初に、認可コードフローの定義を復習します。
- ✓ 次に、API（ユーザーに属する情報を持つリソースサーバー）を起動して、リソースを保護するため、Auth0ダッシュボードに登録します。
- ✓ その後、作成している両方のバージョンのアプリケーションをリファクタリングして、このフローを実装し、アプリケーションをOAuth 2.0クライアントアプリケーションに変換するために必要な、それぞれの手順を学べるようにします。

最後に、ユーザーがサインインできるようにし、ユーザーの代わりに起動したAPIから情報を取得できるアプリケーションを作成します。

認可コードフロー

前の章で紹介したとおり、認可コードフローでは、認可コールバックリクエストから直接トークンを取得する代わりに、アプリケーションはまず認可コードを取得します。そして、アプリケーションは認可コードを認可サーバーの他のエンドポイントへのリクエストに使用し、必要となるトークンと認可コードを交換します。

インプリシットフローと比較した場合のこのフローが持つ最大の利点は、そのセキュリティにあります。具体的には、セキュリティにおいて認可コードフローがよりよい選択であるとする、2つの特徴があります。1つ目は、コードとトークンの交換がバックチャネルで行われることです。つまり、トークンをユーザーのデバイス経由で送信させる代わりに、認可コードフローは、アプリケーションと認可サーバー間で直接開かれているチャネルを使用します。このダイレクトチャネルは、トークンが間違った者の手に渡るリスクを劇的に減少させます。

2つ目の特徴は、トークンを発行する前に、認可サーバーがアプリケーションに認証を要求することです。この認証プロセスは通常、認可サーバーにより割り当てられたクレデンシャルを使用する、アプリケーションで発生します（サインインするために、ユーザーにユーザー名とパスワードを入力してもらうことと似ています）。

要約すると、トークンの交換により信頼できるチャネルを使用しているため、またアプリケーションはトークンの取得前にクレデンシャルを使用して認証するため、アプリケーションと認可サーバーはフローをより信頼することができます。

リソースサーバーを使用する

手短な要約として、リソースサーバーはユーザーに属するリソースを提供するエンティティを参照します。以前にTwitter APIについて読んだ例と全く同じように、通常リソースサーバーは、ユーザー情報（リソース）が安全に保管されている、ユーザー自身または明示的に認可が与えられたサードパーティアプリケーションのみからアクセス可能な、APIを参照します。

このセクションでは、リソースサーバーとして振る舞うAPIを使用します。このAPIは4つのエンドポイントを公開し、ユーザー（またはユーザーのためのアプリケーション）はto-do項目を管理することができます。これらのエンドポイントは各自、ユーザーがアプリケーションに与える必要がある、関連する特定のOAuth 2.0スコープを持ちます。

* **重要な用語:** OAuth 2.0は**スコープ**を使用して、クライアント（アプリケーション）からリソースオーナー（ユーザー）への、特にアプリケーションが希望する動作、またはアプリケーションが代わりにアクセスしたい情報の伝達を可能にしています。この章に先立って、OpenID Connect認可リクエストを使用して、アプリケーションが取得を希望するユーザーの情報を定義しました。少し考えると、適用方法は全く同じであることが分かります。OpenID Connectでは、アプリケーションはスコープを使用してユーザーの代理として使用したいプロファイルデータを定義しました。OAuth 2.0では、アプリケーションはスコープを使用してユーザーの代理として使用したい種類のデータ（または実行したい動作）を定義します。

エンドポイントへのアクセスを制限するためにAPIが使用するスコープは次の通りです。

`read:to-dos` – このスコープにより、アプリケーションがユーザーの代わりにto-do項目を読み込むことが可能になります。

`create:to-dos` – このスコープにより、アプリケーションがユーザーの代わりにto-do項目を作成することが可能になります。

`update:to-dos` – このスコープにより、アプリケーションがユーザーの代わりにto-do項目を更新することが可能になります。

delete:to-dos – このスコープにより、アプリケーションがユーザーの代わりにto-do項目を削除することが可能になります。

APIの役割を知り、スコープについての知識を深めることができたので、設定したAPIのローカル環境での実行をこれから開始できます。始めたばかりの人は、Auth0ダッシュボードのAPIセクションに移動して「Create API」ボタンをクリックする必要があります。それから、Auth0が表示するフォームに次のように入力します。

API Name：**To-Dos API**

API識別子：**https://to-dos.somedomain.com**

署名アルゴリズム：**RS256**

APIに付ける名称はプロトコルの観点からは重要ではありません（これはAPIの内容を簡単に識別するためのみに使用できるフィールドです）。しかし、他の2つは非常に重要です。最初のAPI Identifierは、認可サーバーに、アプリケーションがアクセスする必要があるリソースサーバーを伝えるために使用する値になります。2つ目の値であるRS256は、アクセストークンに署名するために、認可サーバーが使用しなければならないアルゴリズムの種類を定義しています（この例では、非対称アルゴリズムを使用するようAuth0に伝えています）。

* **注：** OAuth 2.0フレームワークで、推奨されるアクセストークンの署名方法を指定しているものはありません。実際、アクセストークンは構造を全く持たないopaqueなアイテムとして考えられるべきです。言い換えれば、アプリケーションはそれらをランダムな文字列ととらえる必要があります。

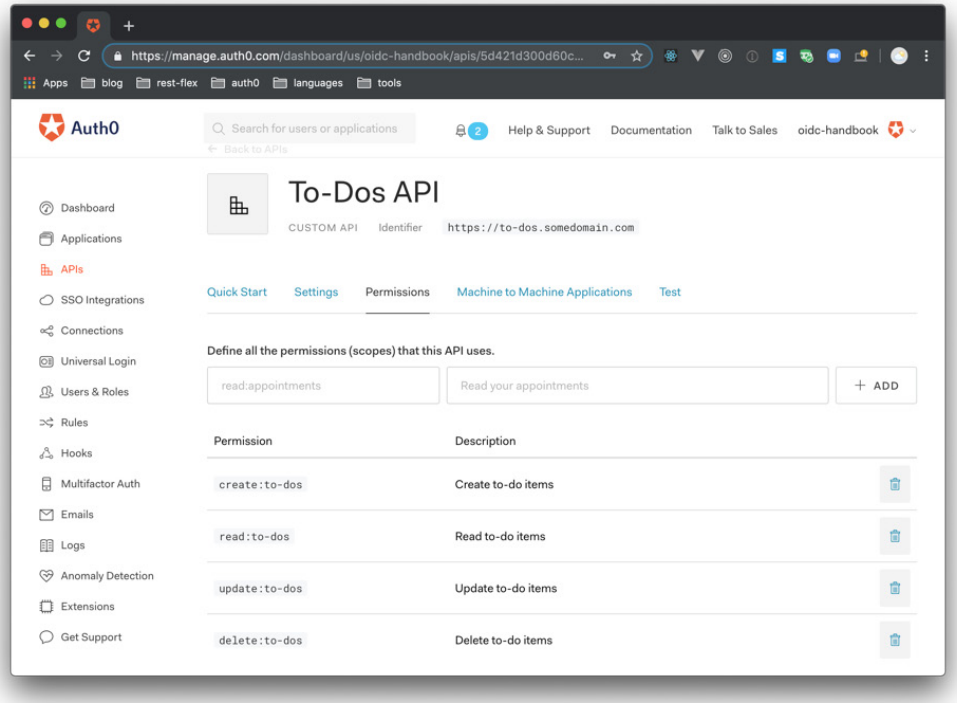
では、「Create」ボタンをクリックしてください。Auth0により新しいAPIの「Quick Start」セクションにリダイレクトされます。そこから、「Permissions」セクションに移動し、フォームを使用して次の4つのスコープを追加します。

read:to-dos – to-do項目の読み込み

create:to-dos – to-do項目の作成

update:to-dos – to-do項目の更新

delete:to-dos – to-do項目の削除



このセクションには保存ボタンがありません。そのため、これらのスコープの追加が終わると、ターミナルを開いてAPIプロジェクトを取得してローカルで実行することができます。ターミナルで、いつもプロジェクトを保存しているディレクトリに移動し、次のコマンドを実行します。

```
git clone https://github.com/auth0-blog/oidc-oauth2-api.git
cd oidc-oauth2-api
npm install
```

最初のコマンドを実行してAPIをクローンする代わりに、GitHubのUIを使用してプロジェクトをダウンロードすることもできます。どちらの方法でも、プロジェクトディレクトリに移動して（2番目のコマンド）、

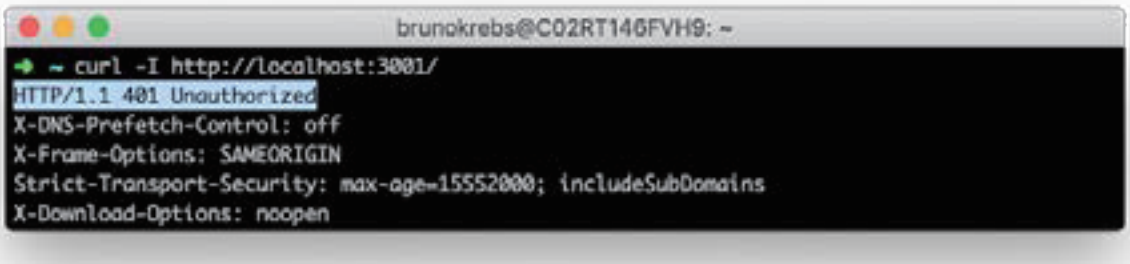
全てのプロジェクトの依存パッケージをインストール（3番目のコマンド）していることを確認する必要があります。それを行った後、.envという新規ファイルをプロジェクトのルートに作成し、次の変数を追加する必要があります。

```
OIDC_PROVIDER=
API_IDENTIFIER=
```

これらは、APIを実行するために必要な2つだけの環境変数になります。最初の変数については、従来のウェブアプリケーションでOIDC_PROVIDER変数に使用した値と、全く同じ値を使用することができます（oidc-handbook.auth0.comなどのもの）。2つ目の変数については、Auth0ダッシュボードで先ほど登録したAPIの識別子を使用する必要があります（https://to-dos.somedomain.comなど）。それらを入力すると、npm startを実行してAPIをトリガーすることができます。それを実行した後、次のようにHTTPリクエストを発行して、APIへのアクセスが本当に制限されていることを確認することができます。

```
curl -I http://localhost:3001/
```

このコマンドを発行すると、APIがHTTP 401 Unauthorizedと応答していることが分かります。この章の最後で、APIがアプリケーションから発行されたリクエストに対し、to-do項目で応答するところを見ます。



委任認可のリクエスト

リソースサーバーの起動が終われば、前の章で構築したアプリケーション（Auth0のSDKを使用していないもの）をリファクタリングし、ユーザーのサインイン時に、To-Do APIに対し委任認可をリクエストするよう変更しましょう。つまり、このセクションでは、/loginエンドポイントをリファクタリングし、認可サーバーに対し、アプリケーションがユーザーの代わりに行いたい動作を伝える、認可コードフローを開始するよう変更します。そして、/callbackエンドポイントを、コードとアクセストークン（アプリにユーザーをサインインさせたい場合はIDトークンも）を交換するようリファクタリングします。

このリファクタリングを開始するには、src/server.jsファイルを開き、/loginエンドポイントの定義に移動して、エンドポイントの最初に定義されている次の3つの定数を変更します。

`responseType`: この定数をcodeに変更して、認可サーバーに対し、アプリケーションは認可コードフローを希望していることを認可リクエストが伝えるよう変更します。

`scope`: アプリケーションがリクエストしているスコープのリストに**`read:to-dos`**を追加します。このスコープにより認可サーバーは、ユーザーの代わりにアプリケーションが行いたい動作の種類を知ることができます。

`responseMode`: この定数を**`query`**に変更して、アプリケーションがクエリーパラメーターとして**`code`**の取得を希望していることを、認証サーバーに伝えます。

* **注：**コードはユーザーの個人情報や他の機密データを含まないため(IDトークンとアクセストークンには含まれています)、クエリーパラメーターとしての取得は問題となりません。誰かがこのコードを傍受しても、トークンと交換するにはアプリケーションのクレデンシャルも必要になります。

これらの定数を変更した後、nonce定数の下に次の定数を追加します。

```
const audience = process.env.API_IDENTIFIER;
```

それから、アプリケーションが**API_IDENTIFIER**オーディエンスへのアクセスを希望していることを認可サーバーに伝えるため、**redirect**関数の呼び出し時に新規定数を使用します。これらの変更により、/loginエンドポイントの定義はこのようになります。

```
app.get(,/login‘, (req, res) => {
 // define constants for the authorization request
 const authorizationEndpoint = oidcProviderInfo[,authorization_
endpoint‘];
 const responseType = ,code‘;
 const scope = ,openid profile email read:to-dos‘;
 const clientID = process.env.CLIENT_ID;
 const redirectUri = ,http://localhost:3000/callback‘;
 const responseMode = ,query‘;
 const nonce = crypto.randomBytes(16).toString(,hex‘);
 const audience = process.env.API_IDENTIFIER;

 // define a signed cookie containing the nonce value
 const options = {
  maxAge: 1000 * 60 * 15,
  httpOnly: true, // The cookie only accessible by the web server
  signed: true // Indicates if the cookie should be signed
 };

 // add cookie to the response and issue a 302 redirecting user
 res
  .cookie(nonceCookie, nonce, options)
  .redirect(
   authorizationEndpoint +
```

```
      ,?response_mode='+ responseMode +
      ,&response_type='+ responseType +
      ,&scope='+ scope +
      ,&client_id='+ clientID +
      ,&redirect_uri='+ redirectUri +
      ,&nonce='+ nonce +
      ,&audience=' + audience
    );
});
```

想像できるように、**API_IDENTIFIER**変数をアプリケーションでまだ定義する必要があります。これを行うには、.envファイルを開き、そこに次の変数を追加します。

```
# … other variables …
API_IDENTIFIER=
```

この変数を設定するには、Auth0ダッシュボードで登録したAPIの識別子を使用する必要があります（APIプロジェクトの**API_IDENTIFIER**で設定したものと同じ値）。

次に、validateIDTokenという新規関数を/loginエンドポイントの下に作成します。この関数は現在のバージョンの**/callback**エンドポイントが行っているものと近い動作を示します。つまり、IDトークンを検証しますが、次のように違った方法で行います。

```
function validateIDToken(idToken, nonce) {
 const decodedToken = jwt.decode(idToken);

 // fetch ID token details
 const {
  nonce: decodedNonce,
  aud: audience,
```

```
    exp: expirationDate,
    iss: issuer
  } = decodedToken;
  const currentTime = Math.floor(Date.now() / 1000);
  const expectedAudience = process.env.CLIENT_ID;

  // validate ID tokens
  if (
    audience !== expectedAudience ||
    decodedNonce !== nonce ||
    expirationDate < currentTime ||
    issuer !== oidcProviderInfo[,issuer‘]
  )
    throw Error();

  // return the decoded token
  return decodedToken;
}
```

この関数のコードを**/callback**のコードと比較すれば、主な違いは、新規関数がIDトークンの署名を検証しないことだと分かります。IDトークンの署名を調べても問題にはなりません。しかし、有益でもありません。重要なのは、新しいバージョンのアプリケーションは、安全なチャネル（TLS証明書を使用）を経由してこれらのトークンを取得するため、アプリが情報の完全性を信頼できることです。IDトークンを検証した後、新規関数はデコードされたバージョンを返すので、アプリケーションでユーザーのプロファイル情報を表示させ続けることができます。

* **注：**アプリケーションはIDトークンの他の項目を、使用する前に調べる必要があります（オーディエンスはアプリケーション自体になっているか、トークンの有効期限が切れていないか、など）。

これでIDトークンを検証する関数が追加されたので、/callbackエンドポイントのリファクタリングができます。それを行うには、このエンドポイント定義を探して、次のように変更します。

```
app.get(,/callback‘, async (req, res) => {
 const { code } = req.query;

 const codeExchangeOptions = {
  grant_type: ,authorization_code‘,
  client_id: process.env.CLIENT_ID,
  client_secret: process.env.CLIENT_SECRET,
  code: code,
  redirect_uri: ,http://localhost:3000/callback‘
 };

 const codeExchangeResponse = await request.post(
  `https://${process.env.OIDC_PROVIDER}/oauth/token`,
  { form: codeExchangeOptions }
 );

 // parse response to get tokens
 const tokens = JSON.parse(codeExchangeResponse);
 req.session.accessToken = tokens.access_token;

 // extract nonce from cookie
 const nonce = req.signedCookies[nonceCookie];
 delete req.signedCookies[nonceCookie];

 try {
```

```
    req.session.decodedIdToken = validateIDToken(tokens.id_token,
nonce);
    req.session.idToken = tokens.id_token;
    res.redirect(,/profile‘);
   } catch (error) {
    res.status(401).send();
   }
});
```

このバージョンの**/callback**エンドポイントと古いものの間には多数の違いがあります。初めての場合、このエンドポイントはこれでHTTP GETリクエスト（POSTの代わりに）を受け入れます。**query**レスポンスモードにより認可サーバーは、自動フォーム送信の代わりにGETリクエストによりユーザーをアプリケーションにリダイレクトさせるようになるため、この変更が必要になります。

他の違いは、IDトークンの検証（ここでは**validateIDToken**関数によって処理されている）に焦点を当てる代わりに、このエンドポイントは認可サーバーより取得したコードを、アプリケーションが必要とするトークン（この例では、IDトークンとアクセストークン）と交換することに、より焦点を当てていることになります。**codeExchangeOptions**定数と**request.post**メソッドの呼び出しで見ることができるように、**/callback**エンドポイントはこの交換を、認可サーバーの**/oauth/token**エンドポイントに、5つのパラメーターでPOSTメソッドを発行することで実行しています。

grant_type：このパラメーターは**authorization_code**の値を使用して、認可サーバーに対しアプリケーションが使用しているフローを明示的にしています。

client_idとclient_secret：これらのパラメーターはアプリケーションのクレデンシャルであるため、認可サーバーは誰が通信してきているのかを知ることができます。

code：これは、認可リクエスト成功時にアプリケーションが受け取るcodeになります。

redirect_uri：このパラメーターは認可リクエストに含まれているため、OAuth 2.0フレームワークはアプリケーションにコード交換リクエストにこれを含めるよう要求します。

コード交換リクエストを発行し、認可サーバーよりレスポンスを取得した後、**/callback**エンドポイントはユーザーセッションにアクセストークンを追加し、**validateIDToken**を呼び出して処理を完了します。最後に、エンドポイントはユーザーを**/profile**に前と同じようにリダイレクトします。

認可コードフローを完了し認可サーバーからトークンを取得した後、最後に行うことは、ユーザーの代わりのリソースの利用（またはアクションの実行）にアクセストークンを使用するかを確認することです。これを見るには、/to-dosエンドポイント定義を検索して、次のように変更します。

```
app.get(,/to-dos‘, async (req, res) => {
 const delegatedRequestOptions = {
  url: ,http://localhost:3001‘,
  headers: {
   Authorization: `Bearer ${req.session.accessToken}`
  }
 };

 try {
  const delegatedResponse = await request(delegatedRequestOptions);
  const toDos = JSON.parse(delegatedResponse);

  res.render(,to-dos‘, {
   toDos,
  });
 } catch (error) {
  res.status(error.statusCode).send(error);
 }
});
```

新しいバージョンのこのエンドポイントが、HTTPリクエストを以前に実行したAPI（**url：‘http://localhost:3001‘**）に、ユーザーのアクセストークン（**req.session.accessToken**）とともに発行

する設定方法に注意してください。具体的には、これらのリクエストはアクセストークンを、**Bearer**プレフィックスを持つAuthorizationヘッダーに付け加えています。このアプローチでトークンは、ベアラトークン使用の仕様に定義されている、ベアラトークンになります。

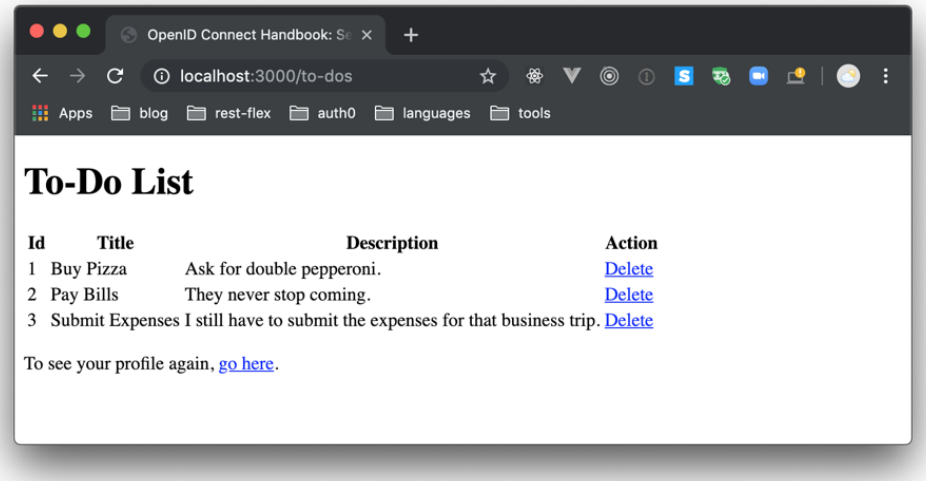
リクエストの発行後、APIがトークンを受け入れた場合（期待される**scope**を含む有効なアクセストークンを取得した時のみ発生）、アプリケーションはリクエストを送ったものは全て取得し、必要となる情報を使用することができます。この例では、アプリケーションはto-do項目のリストを受け取ります（const toDos）。そして、アプリケーションはこの情報を使用してAPIに保存されている全ての項目を持つページを表示します。

アプリケーションの再起動が可能になる前に、これからする必要のある最後のことが、**CLIENT_SECRET**環境変数の定義になります。これを行うには、**.env**ファイルを開き、次の変数を追加します。

```
# … leave the other environment variables untouched ...
CLIENT_SECRET=
```

次に、Auth0ダッシュボードのアプリケーションを開き、Client Secretプロパティをコピーし、それで新しい変数を設定します。他のバージョンのウェブアプリケーション（SDKを使用しているもの）の**.env**ファイルからこの値をコピーすることもできます。

この時点で、ウェブアプリケーションを再起動することで（ターミナルで**Ctrl + C**を入力して**npm start**のコマンドを実行します）、サインイン後に**http://localhost:3000/to-dos**のページにアクセスできます。



このページに表示されている情報が、To-Do APIがアプリケーションに提供している情報になります。つまり、ブラウザに表示されているto-doリストの3つの項目が、アプリケーションがAPIから代わりに取得して利用している情報になります。

* **注：**アプリケーションへのサインインに使用するユーザーが異なっても、To-Do APIが同じデータを返していることに気が付くと思います。本書での手順をシンプルにするため、APIはこのように動作しています。実際のリソースサーバーAPIでは、通常、保持されるデータはユーザーによって異なります。

/to-dosエンドポイントを実装して、アプリケーションをテストした後に実行できる、興味深いことがもう一つあります。練習で**/remove-to-do/:id**エンドポイントの実装を試すことができます。このエンドポイントを実装するコードは/to-dosエンドポイントと非常に近いものになりますが、（ヒントを言うなら）少し違ったアクセストークンが必要になり、リクエストから:idの値を取得する必要があります。

委任認可のリクエストにSDKを使用

前の章の「ユーザー認証にSDKを使用」と同じように、このセクションでもSDKを使用することでどれだけ物事が簡単になるかを見ていきます。実際、Auth0のSDKにより物事が簡単になるので、アプリケーションから委任認可をリクエストさせるには、SDKを使用するプロジェクトに2行のコードを追加しもう一つを変更するだけで構いません。

これの動作を見るには、IDEにAuth0のSDKを読み込み、**src/server.js**ファイルを開きます。このファイルで、**auth0Strategy**定数の定義を検索し、ユーザーのprofileにアクセストークンを追加します。これを行うと、**auth0Strategy**定数は次のようになります。

```
// Configure Passport to use Auth0
const auth0Strategy = new Auth0Strategy(
 {
```

```
    domain: process.env.OIDC_PROVIDER,
    clientID: process.env.CLIENT_ID,
    clientSecret: process.env.CLIENT_SECRET,
    callbackURL: ,http://localhost:3000/callback‘
  },
  (accessToken, refreshToken, extraParams, profile, done) => {
    profile.idToken = extraParams.id_token;
    profile.accessToken = extraParams.access_token;
    return done(null, profile);
  }
);
```

これが、このプロジェクトに追加する必要がある最初の行です。それでは、他の2箇所の変更を行うため、**/login**エンドポイントの定義を検索して、次のように変更します。

```
app.get(
  ,/login‘,
  passport.authenticate(,auth0‘, {
    audience: process.env.API_IDENTIFIER,
    scope: ,openid email profile read:to-dos‘
  })
);
```

ご覧のように、新しい定義では**API_IDENTIFIER**を持つ**audience**パラメーターが**passport.authenticate**の呼び出しで渡されるオブジェクトに追加され、アプリケーションがリクエストするスコープのリストにread:to-dosが追加されています。この変更により、Auth0のSDKは認可サーバーにアプリケーションが必要とするアクセストークンの種類（対象のオーディエンスとスコープ）を伝えられるようになり、アプリケーションはユーザーの代わりにTo-Do APIを利用できるようになります。

次に、この機能を完成させるため、**/to-dos**エンドポイントの定義を入れ替えます。

```
app.get(,/to-dos‘, async (req, res) => {
 const delegatedRequestOptions = {
  url: ,http://localhost:3001‘,
  headers: {
   Authorization: `Bearer ${req.session.passport.user.accessToken}`
  }
 };

 try {
  const delegatedResponse = await request(delegatedRequestOptions);
  const toDos = JSON.parse(delegatedResponse);

  res.render(,to-dos‘, {
   toDos
  });
 } catch (error) {
  res.status(error.statusCode).send(error);
 }
});
```

このエンドポイントの定義とAuth0のSDKを使用しないプロジェクトでの定義の唯一の違いは、アプリケーションがユーザーのアクセストークンを取得する箇所です。他の部分に変更はありません。

新しいバージョンのこのアプリケーションを実行するには、**.env**ファイルを開き、Auth0 APIのIDを渡す**API_IDENTIFIER**変数（前のセクションで使用したのと同じ値）をそこに追加し、**npm start**を実行します。これら少しの変更のみで、以前と同じ結果を実現することができることが分かりました。

まとめ

この章では、委任認可のコンセプトについて深く掘り下げ、アプリケーションがアクセストークンを安全な方法で取得するのに、認可コードフローがどのように役立つかを見ていきました。つまり、バックチャネルとアプリケーションのクレデンシャルを使用することで、このフローはユーザーのデバイスとの間でトークンを送信することを避け、全てのプロセスのセキュリティを強化していることを、ここで学びました。

これらのトピックを抽象的に紹介した後に、API（リソースサーバー）を実行し、OAuth 2.0スコープのコンセプトを深く掘り下げました。次に、両方のバージョンのウェブアプリケーション（Auth0のSDKが無いものとSDKを利用しているもの）をリファクタリングし、認可コードフローの実装方法と、アプリケーションがユーザーの代わりにリソースサーバーを利用する方法を、動作させながら見ました。

少しずつ、OpenID ConnectとOAuth 2.0フレームワークの理解が深まってきたと思います。間もなく、これらの仕様のコンセプトについてより身近に感じ始めるでしょう。また、信頼できるSDKを使用すれば、OpenID ConnectとOAuth 2.0を使用することは、複雑であるにかかわらず、それほど困難ではないことが分かるでしょう。

OpenID Connectと
シングルページアプリケーション

OpenID Connect認証を従来のウェブアプリケーションで扱う方法を学んだ後、自ずと浮かぶ疑問はこうです。シングルページアプリケーション（SPA）で同様のことをどのように実現できるか？ウェブアプリケーション開発の新しい時代において、ほぼ間違いなく、SPAは従来型のウェブアプリケーションより人気があります。ここでは、双方のアプローチにおける長所と短所については触れません。また、コミュニティーではこのトピックについて大きく議論されていますが、ゼロから構築する方法についても触れません。その代わりに、SPAのスケルトンをGitHubから取得して、OpenID Connectを用いてエンドユーザー認証を扱う方法に、焦点を当てます。

このタスクのコーディング作業に取り掛かる前に、認証処理が発生する様子、このフローとこれまでに学んだフローとの主な違い、そしてその違いがある理由について簡単に見ていきます。また、SPAをOIDCプロバイダと統合するため、ダッシュボードにもう一つのAuth0アプリケーションを作成します。

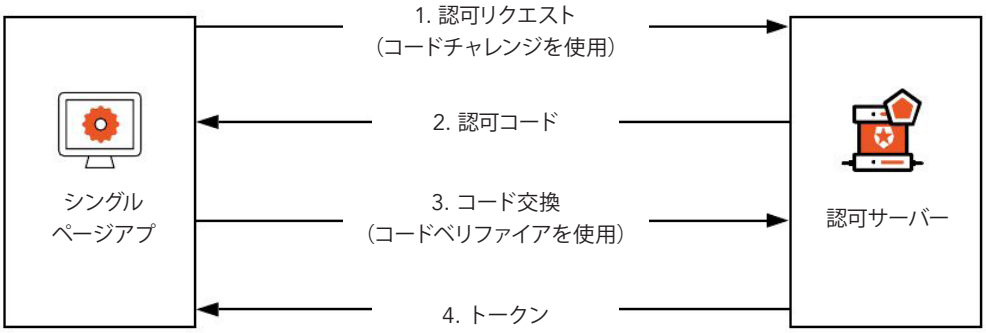
SPAの認証処理

前の2つの章で、次の2つの異なる認証フロー（OAuth 2.0の用語を使えば認可フロー）について学びました。それはフォーム送信によるインプリシットグラントと、認可コードフローです。前者では、エンドユーザーが従来のウェブアプリケーションに認証できるように使用しました。また、後者では同じ結果を実現しつつ、ユーザーの代わりにリソースサーバーとのやり取りに使うアクセストークンをアプリケーションから取得させるように使用しました。

この章では、PKCEによる認可コードフローと呼ばれる、後者の拡張について学びます。PKCE（ピクシーと発音）はProof of Key for Code Exchange（コード交換用証明キー）の頭字をとったものです。名称が

示しているように、このフローはアプリケーションにコード交換のためのキーを持っていることを証明するよう要求します（つまり、フローの最後の段階で、認可コードを提示してアプリケーションが必要とするトークンを取得している間）。これから学んでいきますが、このキーでは、認可コードを使用しているリクエストの送信者が、認可プロセスを前に開始した者と同一であることを証明することが重要です。

以降のセクションで、このキーの生成方法とどのようにこの証明が発生するか、一つずつ見ていきます。簡単に言えば、認可リクエストを発行するとき、アプリケーションでセキュリティを守る必要がある、コードベリファイアから算出されるコードチャレンジと呼ばれるアイテムを送信します。次に、認可サーバーがこのコードチャレンジを、アプリケーションがトークンの取得にコードを送信するまで、保存します。コード交換を発行している間、アプリケーションはコードベリファイアを追加し（認可サーバーがそれをもう知っているため、もうチャレンジではない）、そして認可サーバーは、トークンをリクエストしているエンティティが、認可リクエストに使用されたコードチャレンジを生成したコードベリファイアを知っていることを確認できます。



非推奨のインプリシットグラントの代替手段について

シングルページアプリケーションをOpenID ConnectとOAuth 2.0プロバイダに統合した経験が以前にあれば、（悪い意味でも）有名なインプリシットグラントについて聞いたことがあるかもしれません。OpenID Connectと従来のウェブアプリケーションの章で学んだように、認可リクエストの結果としてトークンを取得する代替手段として、シングルページアプリケーションにはこのフローがあります。つま

り、このフローでは、SPAはコード交換の手順を省略し、認証サーバーがユーザーをリダイレクトして戻すときに、トークンを取得します。

しかし、この代替手段にはセキュリティの観点で問題があります。他の章で学んだプロセスと違い、インプリシットフローでトークンを取得するには、認可サーバーとSPAはURLフラグメントまたはクエリー文字列に頼らざるを得ません。従来のウェブアプリケーションでサインインフローを実装したときのように、フォームの送信を使用することはできません。フラグメントを使用することに決めた場合、トークンはユーザーのブラウザの履歴に記録され、またクエリー文字列に決めた場合は、トークンはSPAがホストされているサーバーに送られるため、このアプローチは問題です。要約すれば、このトークンの取得を意図されていないソフトウェア（ブラウザまたはホストされているサーバー）が、これを見ることができるようになるので、別の攻撃の可能性を生み出すことになります。

この章の目標は、シングルページアプリケーションにOpenID Connectプロバイダを正しい方法で統合することに焦点を当てることにあるので、このアプローチの詳細はこれ以上掘り下げません。それでもトピックの全体を学ぶことに興味がある場合は、リファレンスに使用できるよいリソースとして、Auth0の開発主任であるVittorio Bertocciが書いたOAuth2のインプリシットグラントとSPAの記事をご覧ください。

シングルページアプリケーションをAuth0に登録

抽象的な観点から、シングルページアプリケーションとOpenID Connectプロバイダの統合が機能する方法を学んだので、プロセス全体を動作させながら見ていきましょう。始めたばかりの人は、アプリケーションをOIDCプロバイダに登録する必要があります。これを行うには、ダッシュボードの「Applications」セクションに移動し、「Create Application」ボタンをクリックします。

これを行うときに、次の項目を入力できるフォームが表示されます。

Application Name: OIDCシングルページアプリケーション

Application Type: シングルページウェブアプリケーション

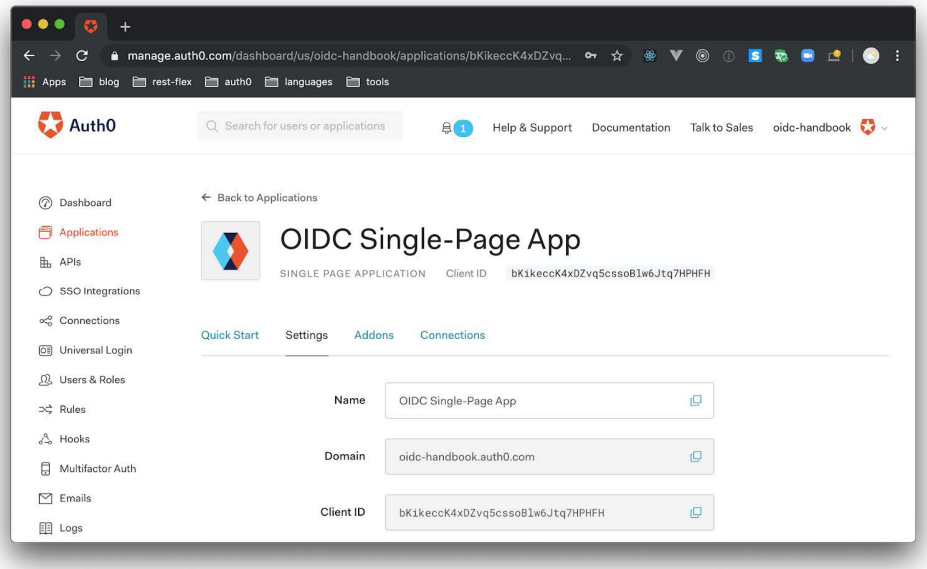
次に、「Create」ボタンをクリックした後、Auth0により新しいアプリケーションの「Quick Start」セクションにリダイレクトされます。そこから、「Settings」タブをクリックして次のように設定します。

AllowedCallbackURLs: http://localhost:3000/#callback

Allowed Web Origins: http://localhost:3000

Allowed Logout URLs: http://localhost:3000

これを入力し、ページの下部にスクロールして、「Save Changes」ボタンをクリックします。Auth0でのSPAの設定はこれで完了ですが、後でいくつかの項目をコピーする必要があるのでこのページを開いておきます。



プロジェクトのブートストラップ

OIDCプロバイダにアプリケーションを登録すれば、新規ターミナルを開いて次のコマンドを実行し、GitHubからプロジェクトのスケルトンを取得することができます（いつものように、ターミナルの代わりにGitHubのウェブサイトを使用してプロジェクトをダウンロードすることもできます）。

```
git clone https://github.com/auth0-blog/oidc-spa.git
```

次に、任意のIDEでプロジェクトを開きます。今回は、NPMプロジェクトではなく、多数の静的ファイルを持つプロジェクトであることが分かると思います。このプロジェクトを実行するには、NPMライブラリをグローバルにインストールして、それを静的なホスティングサービスのシミュレーションに使用します。このライブラリをインストールするには、ターミナルで次のコマンドを実行します。

```
npm install http-server -g
```

プロジェクトに戻ると、2つのディレクトリといくつかのファイルがプロジェクトのルートに作成されています。

`index.html`: これが、静的なホスティングサービスがブラウザに提供するメインとなるHTMLファイルになります。このファイルが、bootstrap.min.css、jwt-decode.min.js、oidc.js、およびapp.jsなどのファイルの読み込みを担います。

`app.js`: このファイルがシングルページアプリケーションのエンジンになります。ご覧のように、このプロジェクトはReact、Angular、およびVueなどの人気のあるフレームワーク（またはライブラリ）を使用していません。そのため、アプリケーションの統合のためコードがいくらか必要になります。このファイルについて詳しく知っておく必要はありません。このファイルがアプリケーションの原動力となることを知っているだけで十分です。

`util.js`: このファイルにはアプリケーションがOIDCプロバイダと統合する必要がある5つの関数が含まれていますが、正確には標準の一部ではありません。たとえば、このファイルにはcreateRandomStringという関数があるのが分かると思います。アプリケーションをプロバイダに統合する間、コードベリファイアの生成にこの関数を使用します。また、これを使用してコー

ドベリファイアをハッシュ化してコードチャレンジとして送信するための、sha256という関数も見つかると思います。そのため、ご覧のように、これらの関数は重要ですが、これはOpenID Connect自体の一部ではありません。

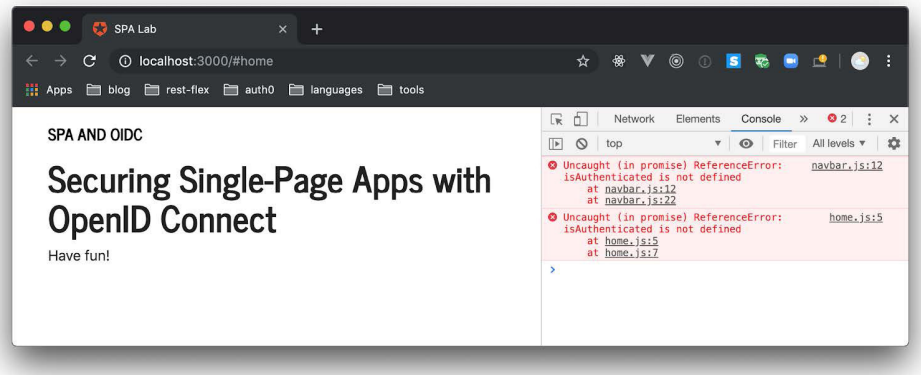
`oidc.js`: これはSPAとOIDCプロバイダの統合のために使用する空のファイルです。全ての仕事はこのファイルで行われます。他は、このファイルに定義する関数に使用するのみです。

`scripts` と `views`: これらのディレクトリには、それぞれのビュー（好みにより、またはコンポーネント）を実行するために読み込まれるコードが含まれています。ディレクトリの内容を確認すれば、それぞれに3つのファイルが見つかります。1つはホームページ、もう1つはto-doリストページ、そして3つ目はナビゲーションバーの表示と制御のためのものです。

この時点で、ターミナルで次のコマンドを実行することができます。

```
http-server . -p 3000 -c-1
```

次に、ブラウザを開いてhttp://localhost:3000に移動すると、ほとんど空の（いくつかのテキスト要素のみ）画面が表示されます。また、デベロッパーツールを開くと、いくつかのエラーが表示されます。



今のところはこのエラーは気にしないでください。この画面が表示されれば問題なく、oidc.jsファイルを実装すれば、このエラーは消えます。

SPAアプリケーションのコードとは別に、このアプリケーションをサポートするto-doリストAPIのブートストラップを行う必要があります。このAPIは前の章で使用したAPIと全く一緒です。そのため、ターミナルを開いてこのAPIのディレクトリに移動し、npm startコマンドを実行してAPIを開始します。すぐにSPAアプリケーションがここにリクエストを発行するので、実行したままにします。

* **注：**APIが利用可能でなくなった場合は、前の章の「リソースサーバーを使用する」のセクションを見てこの後を続ける方法を確認します。APIを実行してそこへのリクエストが拒否されていることを確認すれば、ここに戻って来ます。

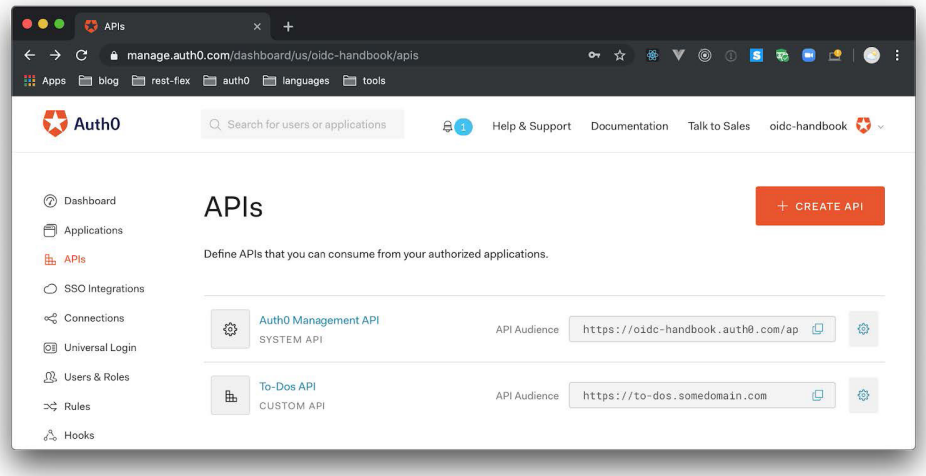
SPAとOIDCプロバイダを統合

OIDCプロバイダにアプリケーションを適切に登録し、マシンでプロジェクトのスケルトンが使用可能であれば、これでシングルページアプリケーションをOpenID Connectプロバイダと統合する方法に焦点を当てる準備は完了です。上述したように、これから開発するコードは全て同じファイル、つまりoidc.jsに入ります。そのため、このファイルを開き、次の定数を追加します。

```
// constants with your own configuration properties
const oidcProvider = '' ;
const clientId = '' ;
const audience = '' ;
```

これらの定数はAuth0アカウントの設定プロパティを表します。正しく設定するには、開いたままにしているAuth0ダッシュボードのページに戻り、次のように使用します。

- ✓ 「Domain」フィールドの値をコピーして、oidcProvider定数の値に使用する。
- ✓ 「Client ID」フィールドの値をコピーして、clientId定数の値に使用する。
- ✓ ダッシュボードのAPIセクションに移動して、「API Audience」フィールドの値をコピーし、audience定数の値に使用する。



その後、oidc.jsファイルに次の定数を追加します。

```
// constants that represent configuration you won't need to change
const responseType = 'code' ;
const redirectURI = 'http://localhost:3000/#callback' ;
const scope = 'openid profile email read:to-dos' ;
const codeChallengeMethod = 'S256' ;
```

ここで追加する定数のほとんどは、すでにご存じのとおりです。responseType定数は認可サーバー（OIDCプロバイダ）に対し、認可コードフローを希望していることを伝えます。redirectURI定数は認可サーバーに、認証フェーズが完了した後、ユーザーをアプリケーションにリダイレクトして戻したい（具体的にはhttp://localhost:3000/#callbackへ）ことを伝えます。scope定数は、アプリケーションはIDトークンを取得のためOpenID Connectの使用（つまり、ただの委託認可のためのOAuth 2.0フローではなく）を希望していて、またアプリケーションはユーザーの代わりにto-do APIを利用できるようアクセストークンを期待していることを表現します。

まだ説明していない唯一の定数がcodeChallengeMethodです。この定数はS256を値に持ち、コードベリファイアの変換に使用した暗号学的ハッシュ関数を認可サーバーに知らせます（この例では仕様で推奨されている、SHA-256）。この定数がどのように使用されるか、またコードベリファイアとコードチャレンジがどのように作成されるかについては、この後すぐ見ていきます。

これらの定数を定義した後、コードベリファイアとコードチャレンジの値を持つ変数を定義します。そのため、まだ同じファイルのままで、次のコードを最後に追加します。

```
// code verifier and challenge for the PKCE flow
let codeVerifier = sessionStorage . getItem ( ‚codeVerifier‘ ) || ‚‘ ;
let codeChallenge = ‚‘ ;
```

このコードが、最初にブラウザのsessionStorageよりcodeVerifierの読み込みを試していることに注意してください。ユーザーがアプリケーションにリダイレクトされて戻ってきたとき、アプリケーションはコード交換の間に生成されたcodeChallengeと同じ値を使用する必要があるため、この処理がされています。しかし、アプリケーションを初めて読み込む場合は、この定数は空になります。

次に、ユーザー関連の情報を持つ3つの変数を追加します。具体的には、idTokenを持つ変数、accessTokenを持つもう一つの変数、およびユーザーのprofileを持つ3番目の変数（idTokenになるがデコードされている）を追加します。

```
// user data
let idToken = ‚‘ ;
let accessToken = ‚‘ ;
let profile = ‚‘ ;
```

これを入力したら、次に行うことは認可リクエストをトリガーする関数の作成です。

```
async function login () {
codeVerifier = createRandomString ();
codeChallenge = bufferToBase64URLEncoded ( await sha256 ( codeVerifier
));
sessionStorage . setItem ( ‚codeVerifier‘ , codeVerifier );
window . location = `https:// ${ oidcProvider } /authorize?` +
  `audience= ${ audience } ` +
  `&scope= ${ scope } ` +
```

```
        `&response_type= ${ responseType } ` +
        `&client_id= ${ clientId } ` +
        `&code_challenge= ${ codeChallenge } ` +
        `&code_challenge_method= ${ codeChallengeMethod } ` +
        `&redirect_uri= ${ redirectURI } ` ;
    }
```

ご覧のように、ユーザーがリダイレクトされるURLを作成しているコードは、前の2章で見てきたものと極めて近いものになります。最大の違いは、code_challengeとcode_challenge_methodのクエリー文字列を追加しているところです。

この関数で注意を引く興味深い点は、コードベリファイアとコードチャレンジがどのように作成されるかです。最初に、createRandomStringを使用してコードベリファイアが生成されます。つまり、ベリファイアはアプリケーションのみが知るただのランダムな文字列です。その後、コードチャレンジがコードベリファイアをSHA-256アルゴリズムでハッシュ化することで作成され、その結果がBase64でエンコードされ認可リクエストに追加されます。

この時点では、コードチャレンジのみが認可サーバーに送信されます。ユーザーがそこに来て認証するとき、認可サーバーがこのチャレンジをユーザーと関連付けます。そしてアプリケーションがコード交換のためのリクエスト（コードベリファイアを含む）を発行したとき、認可サーバーは認可リクエストを開始した同一のアプリケーションが、トークンをリクエストしていることを確認することができます。

* **注：**login関数にnonceパラメーターが無いことにもう気が付きましたか？おそらく気付かれていない方が多いのではないでしょうか。この定数は、公式のSDKを信頼することが重要であることを示すためにのみ省略されています。つまり、OpenID ConnectやOAuth 2.0などは、誤解されやすい多数の手順を含む複雑な仕様です。また、それらは進化し続けており、BCP（Best Common Practices）では、アイデンティティの専門家でない人にとってはコードの更新が重荷となるような、重要な変更がされることがあります。一方、公式のSDKを使用すると、セキュリティアップデートやバグフィックスの取得が格段に簡単になります。

login関数を実装したので、関数が認可リクエストのリダイレクトを処理するコードを書く必要があります。これを行うには、次のコードをログイン関数のすぐ後ろに追加します。

```
async function handleRedirectCallback () {
const queryParams = getQueryParams ();
const code = queryParams . find ( queryParam => ( queryParam . key ===
,code‘ ));
const codeExchangeURL = `https:// ${ oidcProvider } /oauth/token` ;
const codeExchangeFormData = new URLSearchParams ();
codeExchangeFormData . set ( ,grant_type‘ , ,authorization_code‘ );
codeExchangeFormData . set ( ,client_id‘ , clientId );
codeExchangeFormData . set ( ,code_verifier‘ , codeVerifier );
codeExchangeFormData . set ( ,code‘ , code . value );
codeExchangeFormData . set ( ,redirect_uri‘ , redirectURI );
const response = await fetch ( codeExchangeURL , {
method: ,POST‘ ,
mode: ,cors‘ ,
cache: ,no-cache‘ ,
body: codeExchangeFormData ,
});
const responseBody = await response . json ();
accessToken = responseBody . access_token ;
idToken = responseBody . id_token ;
profile = validateIdToken ( idToken );
}
```

この関数では、アプリケーションが最初にしていることが、認可サーバーより送られたcodeの取得であることが分かります。次に、このコードでは、アプリケーションは認可サーバーに送ってトークンを取得する、HTTP POSTリクエストを準備して作成しています。アプリケーションがcodeVerifierを追加す

るのが、このリクエストです。そして、認可サーバーがこのリクエストを取得したとき、そこでは同じ関数（SHA-256）を使用してコードベリファイアがハッシュ化され、前に送られたコードチャレンジと結果が合うか確認されます。答えが正解であれば、認可サーバーはリクエストに応え、トークンをアプリケーションに送り返します。

いつものように、IDトークンを認可サーバーから取得した後、アプリケーションではそれが有効であるかチェックする必要があります。トークンの検証をするには、アプリケーションでは、従来のウェブアプリケーションがトークンを裏のチャネルから取得したときと、全く同じ手順を実行する必要があります。つまり、アプリケーションは署名の検証を省略することができますが（これらのトークンは安全なチャネルで取得されているため）、次のクレームを確認する必要があります。

aud: IDトークンがアプリケーション自体に発行されているか確認するオーディエンス

exp: トークンがまだ有効か確認する有効期限

iss: 発行者がOIDCプロバイダであることを確認

nonce: 反射攻撃を防止

これらの検証を行うには、次の関数をhandleRedirectCallbackの後に追加します。

```
function validateIdToken ( idToken ) {
const decodedToken = jwt_decode ( idToken );
// fetch ID token details
const {
   aud : audience , exp : expirationDate , iss : issuer
} = decodedToken ;
const currentTime = Math . floor ( Date . now () / 1000 );
// validate ID tokens
if (
```

```
 audience !== clientId ||
 expirationDate < currentTime ||
 issuer !== `https:// ${ oidcProvider } /`
) {
throw Error ();
}
// return the decoded token
return decodedToken ;
}
```

ここではそれほど複雑なことはありません。多分予想していたとおり、検証は非常に明快です。IDトークンのクレームがよければ、この関数はデコードされたトークンを返し、アプリケーションはこれをユーザープロファイルとして使用することができます。何かが期待したものと違っていれば、この関数はエラーを発行し処理を止めます。

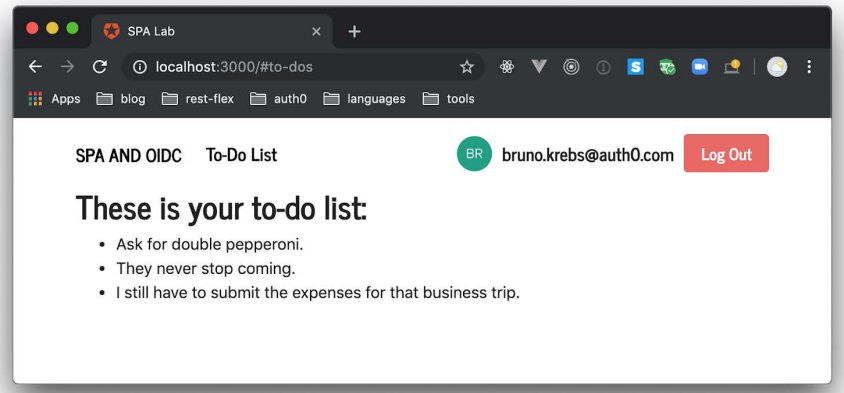
* **注：** login関数がnonce値を認可リクエストとともに送信していないため、validateIdTokenもそれを使用していません。本番環境ではこれをしないでください。実際、このようなコードを最初から作成しないでください。OpenID Connectの機能を学ぶためにここでの情報を用い、SDKを信頼してください。

SPAの実行前にする必要がある最後のことは、アプリケーションの残りが次のようなことをできる関数を追加することです。アクセストークンの使用、ユーザープロファイルの取得、そして認証済みユーザーの有無の確認などです。これを行うには、次のコードをoidc.jsファイルの最後に追加します。それでは、ブラウザのSPAを再読み込みして、エラーメッセージが無くなっていて、ログイン可能で、アプリケーションがto-doリストAPIを代わりに利用できることを確認します。

```
function isAutenticated (){
return idToken;
 }
function getProfile (){
```

```
return profile;
 }
function getAccessToken(){
return accessToke;
 }
```

それでは、ブラウザのSPAを再読み込みして、エラーメッセージが無くなっていて、ログイン可能で、アプリケーションがto-doリストAPIを代わりに利用できることを確認します。



これで終わりです。シングルページアプリケーションにOpenID Connectプロバイダを統合し、その方法を一つずつ学びました。しかしこれほど大変である必要はあるのでしょうか。そうでないことは分かります。次のセクションで、プロフェッショナルなSDKを利用して仕事を代わりにさせ、より安全で簡単な方法で同じ結果を実現する方法について見ていきます。

SPAとOIDCプロバイダを統合 – 簡単な方法

動作する部品のほとんどの知識を得たので、格段に少ない労力で、同じ結果（これから見ていくように、よりよい結果）を実現する方法を学んでいきましょう。始めたばかりの人は、index.htmlファイルを開き、

最後（bodyの終了タグのすぐ前）にある4つのスクリプトタグを探してください。それが見つかれば、最初の3つのタグを削除し、app.jsを含むタグのみ残します。その後、シングルページアプリケーション用の公式Auth0 SDKを読み込むもの、そこでアプリケーションとAuth0を統合する、auth0.jsという新規ファイル（このあとすぐこのファイルを作成します）を読み込むものの、2個のタグを追加します。これらのタグを追加すれば、次のような3つのスクリプトになります。

```
< script
src = „https://cdn.auth0.com/js/auth0-spa-js/1.1.1/auth0-spa-js.
production.js“
>
</ script >
< script src = „auth0.js“ ></ script >
< script src = „app.js“ ></ script >
```

それでは、auth0.jsファイルをプロジェクトのルート（app.jsと同じディレクトリ）に作成し、次の定数を追加します。

```
// constants with your own configuration properties
const oidcProvider = ‚‘ ;
const clientId = ‚‘ ;
const audience = ‚‘ ;
```

これらの定数はoidc.jsファイルで使用したものと全く同じものです。つまり、これらはAuth0の設定を表し、ダッシュボードで使用できるプロパティで設定する必要があります。

これらの定数の設定した後、次の2つをその下に追加します。

```
// constants that represent configuration you won‘t need to change
const redirectURI = ‚http://localhost:3000/#callback‘ ;
const scope = ‚read:to-dos‘ ;
```

上記の定数の両方も、oidc.jsで使用しました。最初のredirectURIは、OIDCプロバイダに認証処理の

後ユーザーをどこに戻すべきか伝えます。もう一つのscopeは、プロバイダにユーザーの代わりにアプリケーションが希望する操作を伝えます（この例では、to-do APIの利用）。

これらを入力したら、最後にある2つの定数の下に次の変数を追加します。

```
// user data
let accessToken = ,‘ ;
let profile = ,‘ ;
```

難しい方法でOIDCプロバイダにSPAを統合したとき、これと似た変数を使用しました。そこでは、idToken（ここではもう使用しません）を保持する変数と、accessTokenを保持するもの、そしてユーザープロファイルを持つ3つ目のものがありました。

ここでは、アクセストークンとユーザープロファイルを持つ変数のみです。公式のSDKが代わりにデコードと検証を処理するので、IDトークンについて心配する必要はありません。

次に、アプリケーションのためのAuth0クライアントを設定する関数を作成する必要があります。

```
async function init () {
 return await createAuth0Client ({
domain: oidcProvider ,
client_id: clientId ,
redirect_uri: redirectURI ,
audience: audience ,
scope: scope
});
}
```

ご覧のように、設定は非常に単純です。公式のSDKが公開するcreateAuth0Client関数を呼び出し、Auth0アプリケーションのプロパティを渡す（audienceとアプリケーションが利用を希望するscopeとともに）必要があるだけです。

Auth0の設定の後、login関数を追加して、ユーザーのサインインを可能にします。

```
async function login () {
 const auth0 = await init ();
 await auth0 . loginWithRedirect ();
}
```

ご覧のように、実装は以前行ったものと比べて格段に簡単です。今回は、コードベリファイア、コードチャレンジ、ノンスの何も気にする必要はありません。Auth0クライアントを初期化して、loginWithRedirectを呼び出す必要があるだけです。

認証コールバックの実装でも、プロセスは同様に簡単です。

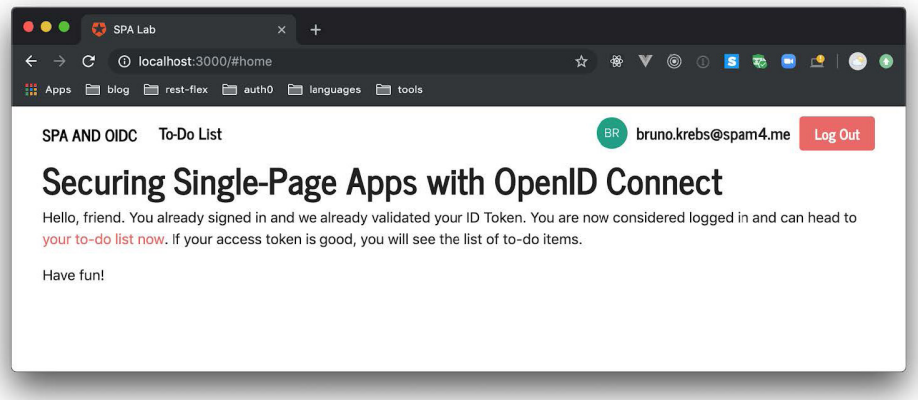
```
async function handleRedirectCallback () {
 const auth0 = await init ();
 await auth0 . handleRedirectCallback ();
 profile = await auth0 . getUser ();
 accessToken = await auth0 . getTokenSilently ();
}
```

この関数では、最初に行うことはAuth0クライアントの初期化で（ユーザーは認可サーバーからリダイレクトされて戻ってきて、古いAuth0クライアントのインスタンスはもう使用できないので、再度実行する必要があります）、次にhandleRedirectCallbackを呼び出します。関数が実行し終えたら、ユーザープロファイルにgetUser関数からアクセスし、アクセストークンをgetTokenSilently関数を使用して取得します。

次に、再度サンプルを動作させるために実行する必要がある最後のことは、SPAアプリケーションがどのように挙動するか（isAuthenticated関数）、またどのようにユーザー情報にアクセスするか（getProfileおよびgetAccessToken関数を使用）の決定を補助する関数を実装することです。これを行うには、次のコードをauth0.jsファイルの最後に追加します。

```
function isAuthenticated () {
 return profile ;
}
function getProfile () {
 return profile ;
}
function getAccessToken () {
 return accessToken ;
}
```

作業はこれで終了です。SPAのAuth0との統合が完了しました。素晴らしかったのは、今回は、コードベリファイアやコードチャレンジの生成や、その他のOpenID Connectの不明瞭な特性を気にする必要がなかったことです。ブラウザでアプリケーションを再度読み込めば、またサインインすることができて、アプリケーションはまだto-doリストAPIを代わりに利用することが可能になります。



まとめ

この章では、シングルページアプリケーションをOpenID Connectプロバイダに統合する方法について学びました。これを行う間、認可プロセスがどのように機能するか抽象的な観点から見て、インプリシットグラントという代替のフローと、それをこれ以上使用すべきでない理由を簡単に見ました。

理論的な部分の後、SDKを利用せずSPAをOIDCプロバイダと統合する方法について一つずつ学んだセクションに入りました。より重要なこととして、手動でのコーディングではどれだけ容易に忘れる（または間違える）かを学びました。

最後に、いくつかの関数を呼び出すだけで、SPAとOpenID Connectプロバイダを格段に安全な方法で統合できることを見ていきました。