文書識別番号:DSP2044

日付:2016 年 06 月 15 日

バージョン:1.0.2

Redfish ホワイト・ペーパー

文書の種類:情報提供

文書の状態:公開

文書の言語:ja-JP

DMTF は、企業やシステムの管理および相互運用性を推進することに力を注いでいる、業界のメンバーから成る非営利団体である。メンバー、およびメンバー以外でも、出典を正しく表示することを条件に、DMTF の仕様と文書を複製することができる。DMTF の仕様は時折改定されることがあるため、特定のバージョンおよび公開日に、常に注意を払う必要がある。

本標準または標準案の特定の要素を実装することは、仮出願済特許権を含む第三者の特許権（本書では「特許権」と呼ぶ）の対象となることもある。DMTF は本標準のユーザーに対し、上記権利の存在について何ら表明するものではなく、上記第三者の特許権、特許権者または主張者の、いずれかまたはすべてを認識、公開、または特定する責任を負わない。また、上記権利、特許権者、主張者の不完全または不正確な特定、公開に対しても責任を負わない。DMTF は、いかなる相手に対して、いかなる方法または環境、またいかなる法論理においても、上記の第三者特許権を認識、公開、または特定しないことに対し何ら責任を負わず、上記第三者の標準に関する信頼性、またはその製品、プロトコル、試験方法論に組み込まれた標準に関しても何ら責任を負わない。DMTF は、上記標準の実装が知見できるか否かにかかわらず、上記標準を実装するいかなる相手に対しても、また、いかなる特許権者または主張者に対しても、何ら責任を負わない。また、DMTF は、公開後に標準が撤回または修正されることにより生じるコストや損失に対し何ら責任を負わず、また、標準を実装するいかなる相手からも、上記実装に対して特許権者が起こす、いずれかまたはすべての侵害の主張から何ら損害を受けず、免責されるものとする。

第三者が保有する特許権であって、DMTF 標準の実装に関連するかまたは影響を与える可能性があると特許権者が考え、すでに DMTF に通知済みである特許権に関する情報については、サイト http://www.dmtf.org/about/policies/disclosures.php を参照のこと。

このドキュメントの標準言語は英語である。英語以外の言語への翻訳が許可されている。

目次

Redfish API とは

Redfish は管理標準であり、データ・モデル表現をハイパーメディア RESTful インタフェースの中で使用する。このモデルは、JSON で表現されているメッセージのペイロードを使用した、標準の機械可読スキーマで表現される。プロトコル自体は、OData v4 を利用している。ハイパーメディア API であるため、一貫性のあるインターフェースを通してさまざまな実装を表すことができる。データ・センターのリソース管理、イベント処理、長期間にわたるタスク、および検出処理のメカニズムを備えている。

1. 新しいインターフェースが必要な理由

様々な影響により、新たな標準管理インターフェースの必要性が発生している。

まず、市場は従来のデータ・センター環境から、スケールアウト・ソリューションへと移行しつつある。スケールアウト・ソリューションとハイパー・スケールでは、普通のサーバーを大量に設置し、中央集中方式ではなく分散方式で一連のタスクを実行する方法を採用してきた。これらの環境での信頼性は、商用あるいはオープンソースのソフトウェアによって得られている。その結果、使用状況のモデルは従来の企業環境の場合とは異なったものになる。類似点は、サーバーをあくまでも道具として使用し、目的とはしないという点だけである。このような顧客は、異種混在でマルチベンダーによる環境でも、一貫性のある、標準に基づいたインターフェースを求めている。

機能指向で均質なインタフェースは、スケールアウト管理の面で不十分である。たとえば、IPMI 機能の使用は、電源オン/オフ/リブート、温度値、テキスト・コンソールなど、コマンドの中でも最小限の共通部分に限定されている。その結果、これらの顧客はアウト・オブ・バンド型の機能を求めながらも、個々のベンダーの拡張機能がすべてのプラットフォームに共通では使用できないことから、不十分な機能セットの使用を強いられてきた。このような新しい顧客たちは、緊密な統合に向けて独自のツール開発を加速し、インバンド型の管理ソフトウェアに頼ることもある。このような環境では管理容易性の共通セットを顧客自身が開発できるからである。OEM 拡張機能の急増に伴ってプラットフォーム管理仕様の断片化が進むと、そのような断片化に起因して、スケールアウトを目指す顧客のニーズに応えられない機能が現れるようになる。また、具体的なセキュリティーと暗号化の観点から見たとき、既存の管理ソリューションでは顧客のセキュリティー要件を満たすことはできなくなっている。

その他の標準（SMASH など）は、期待された遍在性に応えてこなかった。その原因は、それら標準の複雑さにある。CLP は大半のハードウェアに実装されるようになっているが、出力形式に一貫性がなく、構文解析で得られるデータは実装によって異なる。WS Management は、限られた数のアウトオブバンド型の環境でのみ実装されている。WS Management は、単一ベンダによる均質な環境に最も適した複雑な階層化プロトコルであることから、異種環境の要件に対応したことはなかった。また、プロトコル、汎用操作、スキーマ、プロファイルなどをそれぞれ理解して組み合わせることが複雑なので、ソリューションの開発や変更、新機能の追加に数年を要する。重要なリソースのコミットメントやその使用に熟達するための専門知識を理解するにも、数か月の期間が必要である。さらに、単純なタスクであっても、その実行には大量の操作が必要になる。そのため、スケーラブルなシステムになったとしても、インターフェース自体の IO パターンがスケーラブルではない結果となる。

ハードウェアを取り巻く状況は急速に変化している。従来の「ブレード・サーバー」1 台に複数のシステムを置いたシステム、または複数のブレード・サーバーのそれぞれに存在する単一のシステムを集約したシステムのいずれかが一般的になりつつある。これらのシステムは、従来の企業環境を管理しているソフトウェアをそのまま使用して管理する必要がある。一方で、こうした新しいシステムをビット単位のプロトコルで適切に表現することはできない。そのようなプロトコルでは、最新のシステムでコンポーネントどうしが持つ構造上の複雑な関係を表すことができない。さらに、シャーシや筐体の管理機能といったシステムの集約ポイントは、これらのプロトコルを使用したのでは、同様にこのような複雑な機能を実行できない。

最後に、顧客は最新のインターフェースを求めている。顧客は、最近のクラウド・インターフェースで広く普及しているプロトコル、構造、セキュリティー・モデルを使用する API を期待している。これらのクラウド・インターフェースには、顧客がツールに投資して、開発を促進してきたという利点がある。特に、顧客がベンダーからの売り込みによらず自発的に求めているものは、JSON 形式でデータを表現した RESTful な（REST の原則に従った）プロトコルである。

2. REST、JSON および OData を使用する理由

REST は今最も重要なプロトコルとして、急速に SOAP を置き換えつつある。クラウドのエコシステムではその全体で REST を採用していることから、Web API コミュニティーも同様に REST を使用している。また、SOAP よりも習得が容易である。REST は、直接 HTTP 操作にマップするデータ・パターン（厳密にはプロトコルではない）であることによる簡潔性を備えている。

- Web API で広く使用されているプロトコル

REST のセマンティクスは、容易に HTTP/HTTPS にマップできる。そのため、ほとんどのシステム管理者は REST を容易に理解できる。この理由は、すでに広く理解されているセキュリティー・モデルとネットワーク構成の設定を REST が備え、そのサポート方法を把握できるようにしている点にある。また高校やジュニアカレッジでも教えられている。これにより、その使用方法を理解するために社会で必要な教育を大幅に削減できる結果となっている。Web サーバーのオープン・ソース実装は豊富に利用可能であり、プログラミング言語のサポートは豊富である。

JSON は、急速に最新のデータ形式となりつつある。JSON は本質的に人が読める形式であり、XML より簡潔で、最新の各種言語を数多くサポートしているほか、Web サービス API のデータ形式として急速に普及が進んでいる。JSON には、管理の容易性を組み込んだ環境に向けた利点もある。ほとんどの BMC（baseboard management controller）はすでに Web サーバーをサポートしており、ブラウザーを使用したサーバー管理も普通になっている。これは、Java スクリプト駆動インターフェースを通して実行することが普通である。JSON を使用することによって、Web ブラウザーの中で直接 Redfish サービスのデータを使用できる。これにより、ブラウザーで扱うデータと、プログラムによるインターフェースで扱うデータをセマンティクスと値の両面で統一できるようになる。

- 2014 年に広く使用されているコード作成言語
- 新しい API で使用するデータ形式

ただし、RESTful の慣例に従うことと結果を JSON として書式設定することだけでは不十分である。RESTful インターフェースは、アプリケーションとほとんど同じ数で存在しており、公開するリソース、使用可能なヘッダーと照会オプション、結果の表現方法などの面でそれぞれが異なっている。同様に、JSON は理解しやすい表現を提供しているが、共通プロパティ（ID、タイプ、リンクなど）のセマンティクスは、サービスごとに異なる命名規則で適用されている。

OData では、API 間の相互運用性を実現する共通の RESTful 規則群を定義している。

スキーマの記述、URL 規則、および JSON ペイロードの共通プロパティの命名と構造に OData 規則を採用することで、従来の環境とスケーラブルな環境で使用可能な RESTful API のベスト・プ

ラクティスをまとめることができるだけでなく、クライアントの汎用のライブラリー、アプリケーション、およびツールで構成されて規模を拡大し続けるエコシステムで、Redfish サービスの利用を促進できるようになる。

3. ハイパーメディア API を使用する理由

1 つの API スタイルで莫大な数のコンピューティング・プラットフォームに適合できることが必要である。産業界では、複数のインターフェースやプログラミング・スタイルをサポートすることは難しい。スタンド・アロン・サーバーからハイパー・スケール・システム、パーティション化可能システム、さらには仮想システムの市場まで扱う単一のインターフェースが必要とされている。多彩な形態の板金や、その中に収まっているサーバー、および関連する管理機能の間に存在するさまざまな包含関係を示すには、固定 URI の API では不十分である。つまり、1 対多または多対多の関連付けが必要である。

さらに、簡潔なシステムでは取り扱いが容易で、ハイパー・スケールでは柔軟な API であることが必要である。URL を使用して類似リソースの関連性とコレクションを表現する手法は、ハイパーメディア API ではすでに実績がある。

はじめに

Redfish API の容易な理解を図る方法として、運用されている実装を調べることが考えられる。ここからは、実装を調べることによって、アーキテクチャー上のさまざまな構造と概念について説明する。

この手順を始めるには、ブラウザーおよびそのブラウザーからアクセスする実装の 2 つが必要になる。また、ファイル・ブラウザーまたはファイル・マネージャーを使用して、ファイルを調べることもできる。

Redfish 実装は REST と JSON に基づいているので、ブラウザーがあればその実装を確認できる。データを容易に読み込めるように、JSON 形式を表示できるブラウザーの使用を推奨する。ほとんどのブラウザーでプラグインを使用して JSON 形式を読み込むことができる。Chrome Advanced REST クライアントなどの RESTful プラグインをダウンロードすることで、実際の「Redfish」を体験できるようになる。これにより、ヘッダーを設定して、通常はブラウザーで非表示になっている HTTP コードなどの項目を表示できる。この機能は、エミュレーターやベンダー実装のような「実際の」Redfish 実装にアクセスする際にも効果的である。

4. 実装へのアクセス

マークアップは直接表示できるほか、Web サーバーで表示することもできる。実装にアクセスできることだけが必要である。このアクセスを実現するには次のような方法がある。

- パブリックなモックアップへのアクセス：redfish.dmtf.com には JSON を実行している Web サーバーがあり、その Web サーバーをブラウザーで指定すると、目的のデータにアクセスできる。これにより、モックアップとスキーマを調査できるようになる。
- Web サーバーへのデータのコピー：Redfish Mockup にアクセスして、Mock-up ディレクトリー以下にあるすべてのファイルをローカルのハード・ディスクにコピーする。各自のサーバーが HTML ページを提供するために使用する/redfish/v1 ディレクトリーに、コピーしたこれらのデータを配置する。ここに示す例では、nginx をダウンロードし、JSON 形式を返すように設定して、html ディレクトリーにモックアップのファイルをロードする。

5. ルート

Redfish はハイパーメディア API である。したがって、すべてのリソースには他のリソースから返された URL を通じてアクセスする。ただし、どの実装でも開始点が共通になるように、周知の URL は 1 つである。Redfish インターフェースのバージョン 1 では、この URI は「/redfish/v1」である。

URL は、スキーマ（HTTP://の部分）、ノード（www.dmtf.org や 127.0.0.1 のような IP アドレスなど）、およびリソース部分で構成されている。これらをまとめてブラウザーの URL に入力する。したがって、各自のマシン上で nginx サーバーを使用している場合は、ブラウザーに「HTTP://127.0.0.1/redfish/v1/」と入力すると Redfish のルートにアクセスできる。

上記のように構成した URL から、サービスのルートに対して GET を実行する。

基本概念

URL はそれぞれリソースを表している。このリソースとして、サービス、コレクション、エンティティ、その他の構造体が考えられる。なお、RESTful の用語では、このようなURI はリソースおよびそのリソースと相互作用するクライアントを指している。そのため、リソースという用語は、該当の URI にアクセスしたときに返される内容であると考えることができる。

リソースの形式は、スキーマで定義されている。各リソースには、Redfish スキーマで指定された固有のフォーマットがある。このスキーマを使用すると、クライアントはリソースのセマンティクスを判断できる（ここでは、できる限り直感的に記載内容を把握できるようにしている）。Redfish スキーマは、OData スキーマのフォーマットと JSON スキーマのフォーマットで定義されている。OData スキーマのフォーマット（CSDL）で定義されている理由は、汎用の OData のツールとアプリケーションで解釈できるようにすることにある。JSON スキーマ・フォーマットで定義されている理由は、Python スクリプトや JavaScript コードなど OData 以外の環境で解釈できるようにすることおよび見やすさを確保できるようにすることにある。

リソースの構造プロパティは、JavaScript の変数として使用されることを意図している。これにより、適用が加速されるはずであり、JavaScript の Web ページと JavaScript 対応アプリケーションはデータを直接使用できるようになる。リブートを繰り返してもこの URI は持続されるが、クライアントは URI が/redfish/v1/から始まることを想定しているので、URI の検出は/redfish/v1/が起点となる。

よくある間違いとして、URI を固定的に用いることが挙げられる。Redfish はハイパーメディア API であることから、同じベンダーであっても実装によって URI が異なることがある。現在の状態オブジェクトは、目的の状態オブジェクトから分離できる。

6. 操作

操作には、GET、PUT、PATCH、POST、DELETE および HEAD がある。GET は、高度な REST プラグインを使用していない場合に、ブラウザー側で実行される。その他の操作は、エミュレーターと実際の実装でのみサポートされている。

GET はデータを取得する操作である。POST は、リソースの作成やアクション（詳細は後述する）の使用を目的として使用する。DELETE はリソースを削除する操作であるが、現時点で削除できるリソースはわずかである。PATCH は、リソースが持つ 1 つ以上のプロパティを変更するために使用する。一方、PUT はリソース全体を置き換えるために使用する（なお、完全に置き換え可能なリソースはわずかである。詳細は後述する）。HEAD は、ボディデータが返されない点を除いて GET に類似した操作であり、URI 構造を把握する目的で Redfish 実装にアクセスするプログラムで使用される。

7. バージョン管理

Redfish では、2 種類のバージョンを管理する。1 つはプロトコルのバージョンであり、もう 1 つはリソース・スキーマのバージョンである。プロトコルのバージョンは URI に記述されている。先頭が /redfish/v1/となっている理由はここにある。これは、バージョン 1 のプロトコルでアクセスするという意味である。現在利用できるバージョンはバージョン 1 のみであるが、今後現れるであろうバージョンへの対応が必要である。この URI の先頭部分は、実装がバージョン 1 の Redfish 仕様でコンパイルされていることを意味している。この Redfish 仕様は OData v4 に基づいているので、各実装には、値を 4 に設定した OData プロトコル・ヘッダー（OData-Version）を必要とすることにも注意が必要である。

各リソースには、リソース・タイプの定義が記述されている。リソース・タイプは、バージョン管理されている名前空間で定義する。リソースの各インスタンスには、OData タイプの注釈 "@odata.type"を使用して表現したタイプが記述されている。このタイプ注釈の値は、リソース・タイプの URI であり、バージョン管理されている名前空間も記述している。したがって、"@odata.type" : "#ServiceRoot.v1_0_0.ServiceRoot"という表記であれば、ServiceRoot スキーマの 1.0.0 バージョンで定義された ServiceRoot タイプ定義に従うリソースを扱っていることを意味する。それに対応するスキーマ・ファイルは、Redfish スキーマ・リポジトリーの /schema/v1/ServiceRoot に配置されている。したがって、このタイプの完全な URI は "/schema/v1/ServiceRoot#ServiceRoot.V1_0_0.ServiceRoot"になる。スキーマ・ファイルには、リソース・タイプ定義で使用する他のタイプも記述できる（たとえば、構造化されたタイプや列挙）。これらのリソース・パスは同一であるが、各部分では一般に同じ名前空間にある互いに異なるタイプ定義を記述する。

8. 参照

links セクションを調べると、そのセクションに別のリソースへの参照が記述されている。

JSON には、別のリソースを参照するための固有の参照タイプは用意されていない。Redfish は、スキーマについてクライアントが照会する必要がないように、単純な操作では完全に接続されたリソース・ツリーを必要とする。そのため、この仕様では、別の外部リソースまたは内部リソースへの参照を表現するために OData 規則を利用している。

OData の規則に従う他のリソースへの参照を表すプロパティは、その名前の後に"@odata.id"が付記されていることで識別できる。別のタイプの参照（外部ヘルプ・トピックなど）への URL を表すプロパティは、それが URL を表していることを指定する注釈が付記された文字列プロパティとしてメタデータの中で識別できる。

URI は絶対参照または相対参照である。絶対参照は IP アドレスを記述せず、/redfish/v1/から始める。Chrome Advanced REST クライアントのようなプラグインを使用している場合は、それをクリックして、次に実行する GET の URI を得ることができる。

9. 主なオブジェクト

主なオブジェクトは、Systems、Managers および Chassis である。これらは、どれもコレクションである（次の項目を参照）。これらのリソースについて手短に説明する。これらについては少しでも知っておいた方が効果的である。

Systems は、実際に使用している一般的なサーバーを表している。データ・プレーンで CPU からアクセスする対象は、すべてシステムとして表される。これらのシステムは、すべて Systems コレクションに属している。したがって、システムには CPU やメモリーなどのデバイスが存在する。

Managers は、BMC や筐体管理機能など、インフラストラクチャを管理するコンポーネントを表す。Managers はさまざまな管理サービスを扱うが、NIC などの専用デバイスも備えている。

Chassis は、インフラストラクチャの物理的な特性と収容機能を表す。ラック、エンクロージャー、それらに収められたブレードなどは、どれも Chassis である。そのため、Chassis に属する Chassis を表す方法が存在する。シャーシはセンサーやファンなどを収容する。

Systems には 1 つ以上の管理機能を置くことができる（管理機能の中には冗長性を持つものがあるからである）。また、システムは 1 つのシャーシに収められる。Managers は Chassis に収められ、複数のシステムを管理できる。Chassis には複数の Systems と Managers を収めることができる。

以上は概要にすぎないが、ServiceRoot オブジェクトを調べることで、これらはすぐに確認できる。また、Sessions などのサービスがあることもわかる（セッションの詳細は後述する）。

10. コレクション

類似したリソースのグループはコレクションとして表される。モックアップでは、コレクションとして Systems、Managers、Chassis、LogEntries、Sessions、EventSubscriptions などがある。

ここでは、Systems、Managers または Chassis を選択して、その構成内容を調べることにする。これでコレクションの概要を把握できる。

コレクションの応答には、"Members@odata.count"という count プロパティと"Members"オブジェクトが記述されている。"Members"オブジェクトは、メンバーのリストまたはメンバーへのリンクのリストを格納している。メンバーへのリンクは、そのメンバーの URL が記述された単一の"@odata.id"プロパティを格納した JSON オブジェクトとして表される。

コレクションはページ編集されている場合がある。「@odata.nextLink」というプロパティのあるコレクションは不完全なものであり、クライアントは next-link プロパティの URL 値を使用して、このようなコレクションの次の部分をサービスから取得できる。

11. 共通プロパティ

モデル全体を通して、どのような状況でもいくつかの同じプロパティを扱っている場合がある。例えば、どのリソースにも"Name"と"Id"が存在する。"Name"と"Id"は必須のプロパティである。また、埋め込みオブジェクトとして"Status"があり、あらゆる状況下で同じ定義で使用されている。これらすべてのプロパティは、実際にはスキーマの共有部分であり、別のスキーマでも参照することで使用できる。各リソースのスキーマでは、そのスキーマの中で odata "Reference"要素を使用して共通プロパティを参照できる。したがって、どのような状況下でも同じ定義を使用できる。このようなプロパティの例を以下に示す。

- Actions：呼び出し可能なアクションをクライアントに通知する（詳細は、「アクション」セクションを参照）。
- Oem：リソースの標準定義に対するベンダー固有の拡張機能を保持する。詳細は「Oem」セクションを参照。

12. 共通の注釈

注釈であるプロパティも存在する。このようなプロパティは先頭が「@」であるか、プロパティのどこかに「@」が記述されている。注釈として使用できるプロパティには、Odata の注釈と DMTF の注釈の 2 種類がある。OData の注釈は先頭が「@odata.」であるか、注釈のどこかに「@odata.」が記述されている。このようなプロパティの例を以下に示す。

- @odata.type：このリソースを定義しているスキーマの検索に使用する。
- @odata.id：このリソースへの URL を記述する。これは href であるが、Redfish は OData に基づいているので、「href」ではなく「@odata.id」と呼ばれている。
- Members@odata.count：コレクションに存在するリソースの数を定義する。

DMTF の注釈には「@Redfish.」が記述されている。このようなプロパティの例を以下に示す。

- @Redfish.Settings：リソースの設定を示すために使用する（詳細は後述する）。

別の共通の注釈として「@odata.context」がある。これは、実際に汎用の OData v4 クライアントで使用することを目的としたものである。この注釈@odata.context は、OData v4 仕様で適切に定義されているが、基本的には以下の用途で使用する。

1. ペイロードについて記述したメタデータの場所を指定する。
2. 相対参照の解決に使用するルート URL を指定する。

@odata.context の構造は、目的のデータを記述したメタデータ・ドキュメントへの URL である（一般的には、最上位のシングルトンまたはコレクションをルートとする URL）。

技術的には、メタデータ・ドキュメントでは、そのドキュメントで直接使用する任意の型を定義または参照するだけでよい。これにより、各種のペイロードからさまざまなメタデータ・ドキュメントを参照できるようになる。なお、@odata.context では相対参照（@odata.id など）の解決に使用するルート URL が指定されているので、それに適合する「規定的」なメタデータ・ドキュメントを返す必

要がある。さらに、"@odata.type"の注釈には完全な URL ではなく、その一部が記述されているにすぎないので、そのような URL の部分を、該当のメタデータ・ドキュメントで定義または参照する必要がある。また、名前空間のバージョンなし別名でアクションを修飾しているので、参照先メタデータ・ドキュメントでの参照によって、そのような別名も定義しておく必要がある。

たとえば、値"/redfish/v1/$metadata#Systems/Links/Members/$entity"が設定されたプロパティ"@odata.context"がリソース/redfish/v1/Systems/1 にあることが考えられる。これは、汎用 OData v4 クライアントに対し、$metadata で Systems 定義を検索し、Links プロパティ定義を調べることを指示している。このプロパティ定義には、上記のエンティティの定義への参照を記述した Members プロパティがある。

また、"@Redfish.Copyright"という注釈もある。実装は、このプロパティを返すことはない。このプロパティは、モックアップで使用される静的な応答例の著作権声明としてのみ存在している。

13. アクション

REST を使用しても、容易には実行できないものもある。そのために、アクションが用意されている。System の押しボタンのようなもの（設定に応じてシステムのリセットやシャットダウンを実行する機能）を System で表現することは容易ではない。このサービスには、ボタンの状態を扱う考え方がなく、プロパティとして用意されていないからである。アクションの別の用途として、ファームウェアのアップグレードやグレースフル・シャットダウンなど、一定の時間を要する操作がある。こうした操作はアトミック・アクションとして容易に表現でき、クライアントで便利に使用できる。

PUT/PATCH 操作の対象にできる隠しプロパティや複雑な状態マシンの代わりに考え出されたものがアクションである。アクションは、POST 操作を使用してリソースを対象に実行する（詳細は該当の仕様を参照）。どのリソースでも、「Actions」プロパティを調べればサポートされているアクションがわかる。

14. スキーマ

スキーマには、クライアントが使用できる情報が大量に格納されている。クライアントでは、プロパティを中心としたセマンティクスを判断するためにスキーマを調べることが推奨されている。特にプログラム動作のクライアントではこの点が重要である。スキーマには、データ型、列挙型のリスト、プロパティに関する非規定的な情報の記述、プロパティの動作が従うべき規定的な情報などが記述されている。

その他の概念

ここまでは、Redfish API の最も簡潔な形式についての説明である。データがかなり読みやすいことはわかるが、セキュリティー、スキーマの種類、ヘッダーの内容などがどのようになっているかについても知る必要がある。

15. Sessions

セッションを確立せずにアクセスできる対象は、普通はルートのみである。なお、パブリック・モックアップまたは各自のローカル Web サーバーを使用している場合は、セキュリティーが問題にならないのでセッションを確立する必要はない。セキュア・モードで実行しているエミュレーターがある場合や実際の実装で作業している場合は、セッションの確立が必要になる。

まず、実装で有効なユーザー名とパスワードがわかっている必要がある。

セッションとして確立する URI は、セキュリティーで保護していない POST 操作でも使用してセッションを確立できる。この URI を確認するには、Links の Sessions プロパティを調べて、サービス・ルート（/redfish/v1/）の@odata.id プロパティの値を検索すればいいが、ほとんどの場合、この URI は/redfish/v1/Sessions である。

セッションを作成するには、/redfish/v1#/Links/Sessions/@odata.id で示された URI に対し、以下の POST ボディを指定した HTTP POST 操作を実行する。

```
POST /redfish/v1/SessionService/Sessions HTTP/1.1
Host: <host-path>
Content-Type: application/json; charset=utf-8
Content-Length: <computed-length>
Accept: application/json
OData-Version: 4.0

{
   "UserName": "<username>",
   "Password": "<password>"
}
```

この戻り値には、セッション・トークンを記述した X-Auth-Token ヘッダーと Location ヘッダーがある。

戻される JSON ボディには、新たに作成したセッション・オブジェクトの表現が記述されている。

```
Location: /redfish/v1/SessionService/Sessions/1
X-Auth-Token: <session-auth-token>
```

```
{
   "@odata.context": "/redfish/v1/$metadata#SessionService/Sessions/$entity",
   "@odata.id": "/redfish/v1/SessionService/Sessions/1",
   "@odata.type": "#Session.v1_0_0.Session",
   "Id": "1",
   "Name": "User Session",
   "Description": "User Session",
   "UserName": "<username>"
}
```

応答の X-Auth-Token ヘッダーに記述されたトークン文字列は、これから発行するすべてのサービスに配置する同じヘッダーで使用する。セッションを削除するには、応答の@odata.id で返された URL（上記の例では、/redfish/v1/Sessions/Administrator1）に対して DELETE 操作を実行する。

16. 冗長性

いずれかの Chassis に戻り、"Thermal"へのリンクをたどってファンについて調べると、Redfish が冗長性を示す様子を知ることができる。

"Redundancy"という配列があることがわかる。この配列は、RelatedItem プロパティに同じ値を使用している 2 台のファンがあることを示している。Redfish では冗長性に共通のスキーマ定義があり、メンバー以外にも冗長性に関するその他の重要な属性を示すプロパティがある。@odata.id の値は冗長性セットのメンバーへのポインターであることから、各冗長セットが属している項目をクライアントから確認できる。

なお、"@odata.id"プロパティの値がリソース全体を指していないこともある。このプロパティの値の形式は、JSON ポインターまたは OData 参照のいずれかである。

- JSON ポインターの場合は、リソースの終了点とプロパティ・パターンの開始点を示す#が記述されている。このスキーマには、プロパティへの参照も記述されている。JSON ポインターの値は、"/redfish/v1/Chassis/1/Thermal#/Fans/0"のような形式である。
- OData 参照の場合は#が記述されていない。このスキーマには、プロパティの定義が記述されている。JSON ポインターの値は、"/redfish/v1/Chassis/1/Thermal/Fans/0"のような形式である。

17. RelatedItem

さらに"Thermal"リソースを調べると、Temperature 配列の要素が、"@odata.id"を記述した値を持つ RelatedItem プロパティを持っていることがわかる。このプロパティは、この温度センサーの測定対象となっているサブリソースへの参照であり、多くの場合、このサブリソースはプロセッサーである。Redudancy のリンクと同様に、その値はサブリソースを指している。これにより、この温度センサーの測定対象をクライアントから判断できる。

RelatedItem には共通のスキーマ定義がある一方で、その使用方法は状況に依存する。それでも、2 つのリソースどうしまたはサブリソースどうしの関係を示すために使用するという点は変わらない。

また、冗長性と同様に、"@odata.id"プロパティの値がリソース全体を指していないこともある。

18. サービス

ルートに戻り、いくつかの共通サービスについて調べることにする。Tasks、Sessions、EventService および AccountService は、すべて共通サービスである。これらの詳細は仕様で確認できるが、どのようなサービスを提供するかは、その名前から明らかである。Tasks には、通常、アクションの結果として開始されているジョブのリストが格納されている。Sessions については上記で説明済みである。AccountService は、ユーザーを作成する手段である。EventService の詳細については後述する。

19. レジストリー

メッセージ・レジストリーを使用すると、イベントとエラーに使用する短い識別子を管理プロセッサーで保持して、クライアントがレジストリを検索して実際のメッセージを確認することができる。メッセージ単位でパラメーターを置き換えるメカニズムが用意されている。

メッセージ・レジストリーを互いに独立したエンティティーにすると、国際化対応が容易になる。これは、管理プロセッサーでは翻訳した情報を扱う必要がないからである。レジストリー自体を翻訳しておくことで、クライアントは必要な翻訳済み情報を表示できる。

20. ヘッダー

Redfish サービスは、共通の HTTP ヘッダーの一部に対応している。これらすべてのヘッダーは仕様に一覧で記載されているが、最も重要な 2 つのヘッダーとして、Accept ヘッダー（'application/json'に設定する必要がある）および前述の X-Auth-Token ヘッダーがある。実際の実装には、静的なモックアップには見られない多くのヘッダーが存在する。

21. ETag

ETag は、IO（キャッシング）の最適化を目的としてブラウザーで使用する。ETag では If-Match を実行し、一致が得られた場合はデータを引き出す手間を省く。オブジェクトが変更されているかどうかを判断するために ETag を使用する。そのため、PUT を実行した場合または何らかのツール（コンソールでの BIOS 構成など）を使用した場合は各 ETag が変化する。これにより、PUT の実行によって Redfish クライアントと非 Redfish クライアント（または別の Redfish クライアント）の間に発生した競合状態を必ず検知できる。

モックアップでは、実際の Web サービスではないので、ETag の動作は確認できない。

21.1. クライアントの競合状態の処理

競合するクライアントが互いに設定データを上書きしようとする可能性がある。この状態は、ETag と If-Match を使用して処理する。

クライアントの読み取り/変更/書き込みのサイクルは、以下のように構成されている。

1. リソースと共に現在の ETag を取得する。
2. リソースのプロパティを変更する。
3. 新しい ETag を生成する。
4. 内容がわかっている最後の ETag を収めた If-Match ヘッダーを含め、変更済みのリソースに対して PATCH を実行する。
5. 得られた ETag がオブジェクトと一致しなくなっている（読み取り以後に何らかの変更があった）場合は、サービスから 304 Not Modified が返される。クライアントは、上記の手順を繰り返して復旧処理を実行する必要がある。
6. プロバイダーが効率的に変更を検出して、複数のクライアントの競合状態を防止できるようにするには、設定データの更新時にクライアントが ETag を指定する必要がある。

22. リソースの更新

GET と PUT/PATCH による簡潔なメソッドは、クライアントが現行の構成を GET で取得し、同じリソース URI に対して新しい構成で PUT または PATCH を実行するモデルである。PUT または PATCH が成功したことは、HTTP 状況コードと応答ボディで判断する。

これは、組み込みプロバイダーとの相互作用を意図している。これにはアクションと構成の変更をファームウェアで処理して適切に応答できる範囲での Redfish サービス自身のデータなどがある。

PUT または PATCH を実行して得られる HTTP 応答には、HTTP 状況コードが記述されるほか、処理が失敗している場合は、応答ボディとして Extended Error Information 構造体が記述されている。

22.1. PUT と PATCH

PUT と PATCH の両方が存在する理由について説明する。リソースの部分的な置き換えではなく、全体の置き換えを実行するとセマンティクスに問題が発生することが、この仕様の設計時に判明した。リソースの中には、部分的なプロパティの置き換えとリソース全体の置き換えが可能なものがある。リソースに存在する SettingData などのプロパティを消去するべき時機を判断する方法が必要であった。きわめて入り組んだ状況になり始めたことから、1 回の操作で目的を達成することを試すために特殊な（不自然な）規則も設定された。

この問題は PATCH で解決することができた。PUT は必ず全体を置き換え、PATCH は必ず部分

的な置換を実行する。これにより、クライアント側でもサーバー側でも複雑なロジックを必要とせずに、サービスの決定論的な動作を実現できる。

PATCH は業界で広く採用されつつある。Open Stack をはじめとする多数の API が対応済みであり、多くの既製の Web サーバーで利用可能である。

22.2. Current Configurations と Settings

Redfish には、基本的に 2 種類のオブジェクトとして Current Configurations と Settings がある。ほとんどのオブジェクトは、特定のリソースの現在の状態を表している。リソースに"@Redfish.Settings"というプロパティが見られることがある。この注釈には、構成のために PUT 操作や PATCH 操作を実行する条件を指示するリンクが記述されている。このリンクは、リソースの将来の状態を表す。

指定された変更をすぐに処理できるリソースもあれば、変更を有効にするためにシステムやサービスの再起動やリブートを必要とするリソースもある。"@Redfish.Settings"は、リソースの種類とその変更対象箇所をクライアントに通知する目的で使用する。

"@Redfish.Settings"プロパティがある場合、そこには次回のリセットやリブートなどの際に変更が適用されるリソースへのリンクが記述されている。Settings リンクがない場合、なんのタスクも生成されていなければ、オブジェクトに対するどのような PATCH 操作も直ちに実行される。

また、最後に設定がリソースに適用された時期に関する情報も存在する。この情報として、その適用時刻、ETag、その設定が「ライブ」で適用可能であった場合に返されたメッセージなどがある。

Settings が必要になる可能性のあるリソースの例として、BIOS のほか、NIC やストレージなどのデバイスがある。

22.3. 更新可能なプロパティの判断

どのプロパティを更新可能にするかは、スキーマおよびメタデータの両方で定義する。"OData.Permissions/Read"が指定されたプロパティや、read-only が true に設定されたプロパティは更新できない。一方、"OData.Permissions/ReadWrite"または read-only = false が指定されたプロパティは書き込み可能である。どのプロパティがどの条件下で書き込み可能であるかを LongDescription（規定的なテキスト）で示すことができる。既定では、"Permissions"注釈のないメタデータや readonly = true が指定されていないスキーマは書き込み可能であると見なされる。

書き込み可能の指定があったとしても、実装側でプロパティを読み取り専用にすることができるので、そのプロパティが必ずしも書き込み可能であるとは限らない。

複合型プロパティに対するアクセス許可は、その複合型プロパティを構成するプロパティのうち、"Permissions"注釈がないすべてのプロパティに対する既定のアクセス許可と同一になる。たとえば、IPv4Address 複合型を構成する SubnetMask プロパティには"Permissions"注釈が存在しない。EthernetInterface の IPv4Addresses プロパティは、IPv4Address 型である。IPv4Addresses に "OData.Permission/Read" 注釈があると、SubnetMask は読み取り専用になる。

IPv4Addresses に"OData.Permission/ReadWrite"注釈がなければ、SubnetMask は読み取り/書き込み可能である。この点を明確にするためのベスト・プラクティスとして、複合型のすべてのプロパティに"Permissions"注釈を定義する。

Redfish の PUT/PATCH 操作のセマンティクスでは、書き込み不可のプロパティを更新しようとする操作はエラーとして扱われない。Redfish サービスの実装では、更新できなかったプロパティを示す Extended Error Information と共に 200 が返される。

23. Extended Error 応答

HTTP エラーのセマンティクスだけでは、ユーザーやアプリケーションがエラーの原因を把握して修正措置を講じるために必要な情報を十分に提供できない。Extended Error 応答構造体の概念が導入された理由はこの点にある。ExtendedError はレジストリーも扱うことができ、エラーのセマンティクスがクライアントにとってより意味のあるものになる。

たとえば、クライアントが一度に変更する必要のあるプロパティが数十個あるが、そのうちの 1 つが無効なプロパティであるものとする。実装が"400"を返したとしても、エラーのあるプロパティとエラーの理由をクライアントが特定するうえでは役に立たない。また、"200"を返したとしても、いずれかのプロパティが存在しない場合、その応答には意味がない。

この応答に JSON ボディと Extended Error 応答が付随していれば、返されたコードの理由だけでなく、プロパティ名とその他の情報もこの構造体に記述できる。また、ExtendedError にはメッセージの配列も設定できるので、複数の問題が発生したときに複数のメッセージを返すことができる。

24. イベント生成

SNMP や IPMI などのプロトコルに準拠するには、何らかのイベント生成メカニズムが必要になる。このメカニズムは、すでに BMC に存在している。その手法は、サブスクリプションに基づく PUSH メカニズムである。PUSH のイベント生成は BMC に適している。これは、イベントを送信した後はメモリーにそのイベントが維持されないからである。

まず、イベントの送信先になる Events を要求する URI をクライアントが決定する必要がある。POST 要求を EventSubscription コレクションに送信することでセッションを確立できる。

Redfish では、2 種類のイベントとしてライフサイクル・イベントとアラート・イベントをサポートしている。EventService を調べることで、実装でサポートされているイベント生成の種類を確認できる。

イベント生成の詳細については、該当の仕様を参照すること。

25. ユーザー・アカウントなどのリソースの作成

Eventing や Sessions と同様に、リソースは POST 操作で作成する。これは、コレクションに対する POST 操作であることが普通である。イベントのサブスクリプション、セッションおよびアカウント・サービスのいずれにもコレクションがある。これらのコレクションの URI は、このハイパーメディア API の他のリソースと同様に検出される。

このスキーマでは"RequiredOnCreate"という注釈を定義する。クライアントは、この注釈を使用して POST に指定する必要のあるプロパティを判断できる。

26. メッセージの形式合せ

アーキテクトが Redfish での実現を目指す処理として、すべてのメッセージ形式を合わせることが挙げられる。この形式合せでは、イベント・メッセージ、Extended Error 応答メッセージ、ログ・エントリーおよびメッセージ・レジストリーのすべてが対象となる。ベンダー固有の形式や他の形式を実装に保持することもできるが、メッセージ形式を合わせる手段も用意されている。これにより、メッセージを国際化できるだけでなく、ストレージの最適化も実現できる。また、このデータを利用および提示するクライアント・タスクとサービス・タスクの両方でメッセージの統一を図ることもできる。

27. Oem

モックアップでも、また多くの場合は実装でも、"Oem"という構造体があることがわかる。この構造体は、リソースの基本プロパティ、"Links"セクションまたは"Actions"オブジェクトで使用できる。Redfish では、ベンダーが独自のオブジェクトを"Oem"オブジェクトに追加することで、そのベンダー独自の拡張機能を組み込めるようにするメカニズムを定義している。この独自のオブジェクトのスキーマ・ファイルをクライアントから探し出すことができるように、そのオブジェクトには"@odata.type"プロパティを設定する必要がある。この標準では、これを超える要件は設定していない。このメカニズムでは、標準のリソースと同じ構造体を使用した複数の拡張機能を複数ベンダー環境に配置できる。これにより、余分な IO が不要になり、スケーラビリティが向上する。ベンダーには、今後標準に取り入れられるような機能を導入することが望まれる。

28. べき等

どのような RESTful サービスに対しても有効な RESTful 操作は、GET、PUT/PATCH、POST、および DELETE である。

べき等は、Redfish の簡潔化にとって重要な概念である。べき等の操作では、1 回実行した後で操作を繰り返し呼び出しても結果が変化することはない。たとえば、DVD リモコンの「停止」ボタンはべき等である。このボタンを 1 回押すと動画の再生が停止するが、以降はこのボタンを何回押しても動画は停止したままである。一方、「一時停止」ボタンはべき等ではない。このボタンの 2 回目の操作では再生が再開される。非べき等の操作による結果は直前の状態に応じて異なる。

HTTP PUT と PATCH はべき等の操作である。クライアントが 2 回目に同じデータを PUT しても、リソースは変更されない。PATCH 操作ではプロパティが置き換えられるが、PATCH 操作を繰り返してもリソースは変更されない。これは、1 回目の PATCH によりすでにリソースが目的の状態になっているためである。

リソースに対する PUT 操作は、構成を変更するための最も単純な表現である。これは、「リソース X を Y に置き換える」操作を意味する。それ以外の対話パターンは、これよりも複雑になる。PATCH は、リソースの差分更新が必要になるので、やや複雑な操作である。べき等の PATCH 操作を使用して完全に構成できるサービスが理想的である。

Redfish では、基本的な REST 操作を使用する 3 つの基本対話パターンを定義している。

28.1. べき等による変更：GET/PUT/PATCH

リソースに対する PATCH 操作では、指定したプロパティの新しい値でリソースの内容が置き換えられる。この操作は、サービスが所要の構成を利用して結果の状態を生成できるだけである場合に使用する。多くの場合、クライアントはリソースを GET で取得し、プロパティの値を変更したうえで PATCH を実行し、変更内容をコミットする。

リソースに対する PUT 操作では、リソースの内容が新しい値に置き換えられる。この操作は、クライアントが単にリソースを目的の状態に設定しようとする場合に使用する。

28.2. 作成、使用、削除：POST/GET/DELETE（非べき等）

リソースに対する POST 操作では、子リソースの新しいインスタンスが作成される。項目の作成と削除を必要とするデータ・モデルにこの操作を使用することが普通である。具体的な操作として、新規ユーザー・アカウントの作成、ユーザー・アカウントの削除、新規ログ・エントリーの作成、ライセンスの追加や削除などがある。これらの項目は、削除にも対応している。コンテンツを URI に対して POST 処理することで作成が実行される。新しく作成したリソースは、元の URI の子になり、その新しい URI に対する DELETE 操作で削除できる。

28.3. アクションの実行："Action"プロパティを指定した POST（非べき等）

リソースの Actions プロパティに返されたリソースに対して POST 操作を実行するとカスタム・アクションが実行される。同じ指定で POST を繰り返し実行すると追加の動作が実行されるので、このパターンはべき等ではない。カスタム・アクションの例としてテスト・イベントの送信やログの消去がある。

29. 継承とポリモーフィズム

語られることが少ない事項として継承とポリモーフィズムがある。
オブジェクトは、Resourceという1つのマスター・オブジェクトを継承する。Redfishでは、それ以外の継承は使用されない。同様に、Redfishでは共通のプロパティを取得して、それを親オブジェクトに置くこともない。そのため、任意のスキーマから特殊化できるサブオブジェクトは存在しない。つまり、同一のコレクションに親タイプと共にサブタイプが返されることはない。

30. コピーによる継承

Redfishでは共通の定義をとる。その定義がResourceに存在しない場合、作成者は必要なオブジェクトを別のスキーマから切り取って貼り付けることで、その定義をオブジェクトに追加する。この方法では、オブジェクトどうしの同期状態を維持する手間が作成者に発生するが、閲覧者にとっては、特定のオブジェクト、プロパティまたはアクションのセマンティクスを知るだけの目的で別の参照の中を調べる必要がなくなる。

31. ユニオンによるポリモーフィズム

Redfish では、共通のセマンティクスを持つリソースをとり、その定義を同じリソースにすべて配置する。この方法により、類似のリソースには類似のセマンティクスが存在するようになる。実装担当者は、サポートされていないプロパティを使用しないようにするだけでよい。たとえば、類似点が85%のデバイスが3台ある場合は、デバイスどうしの相違となる15%を共通スキーマの定義に追加する。
実装担当者は、リソースに必要な 85%の部分と、実装するデバイスに適用する 15%の定義を使用する。これは、規模の大小に関係なく、あらゆるスキーマで効果がある。このアプローチを検討すると、多くの場合、スキーマとリソースに追加の「リソース・タイプ」形式のプロパティが存在することがわかる。このようなプロパティは、目的のデバイスがどのようなものであるかをクライアントが知るうえで効果的である。

結論

Redfish API は、IT の新しいプログラミング・スタイルの象徴であり、ハイパースケールからブレード、スタンド・アロン・サーバーに至るシステムを一貫した方法で管理する能力を備えている。Redfish API は、顧客の要求と顧客が使用しているツールに適合する自然な進化であると考えられる。

一般的なユースケース

ここでは、アーキテクチャーの理解とクライアント・コードの作成着手に効果的ないくつかのユースケースを示す。

32. システムの温度センサーの検出

アプリケーション・コードは必ずルートの/redfish/v1/から起動する。

1. ルート・オブジェクトが"Systems"というプロパティの場合。
    1. "@odata.id"の値を探し出す。これは Systems コレクションの URI である。
    2. モックアップの場合、この URI は/redfish/v1/Systems である。
    3. その URI に対して GET を実行する。
2. "Members"配列で"Links"オブジェクトを調べる。
    1. 目的とするシステムの"@odata.id"を探し出す（正しいシステムを判断するために、システムごとに GET 操作を実行して、そのシステムにあるプロパティを調べる必要がある）。
    2. モックアップの場合、この URI は/redfish/v1/Systems/1 である。
    3. その URI に対して GET を実行する。
3. "Links"オブジェクトで"Chassis"オブジェクトを調べる。
    1. "@odata.id"の値を探し出す。これは、このシステムが収容されているシャーシである。
    2. モックアップの場合、この URI は/redfish/v1/Chassis/1 である。
    3. その URI に対して GET を実行する。
4. "Links"セクションで"Thermal"オブジェクトを調べる。
    1. "@odata.id"の値を探し出す。これは温度センサーである。
    2. モックアップの場合、この URI は/redfish/v1/chassis/1/Thermal である。
    3. そのリソースに対して GET 操作を実行する。
5. "Temperature"配列を調べる。これはシステムの温度センサーである。
    1. "RelatedItem"を調べて、それぞれの温度センサーが監視している具体的なコンポーネントを特定する。

33. Chassis に含まれる Chassis

アプリケーション・コードは必ずルートの/redfish/v1/から起動する。

1. ルート・オブジェクトが"Chassis"というプロパティの場合。
    1. "@odata.id"の値を探し出す。これは Chassis コレクションの URI である。
    2. モックアップの場合、この URI は/redfish/v1/Chassis である。
    3. その URI に対して GET を実行する。
2. "Members"配列で"Links"オブジェクトを調べる。
    1. 目的とするシャーシの"@odata.id"を探し出す（正しいシステムを判断するために、システムごとに GET 操作を実行して、その Chassis にあるプロパティを調べる必要がある）。
    2. モックアップの場合、この URI は/redfish/v1/Chassis/Enc1 である。
    3. その URI に対して GET を実行する。
3. "Links"オブジェクトで"Contains"オブジェクトを調べる。
    1. "@odata.id"の値を探し出す。これは、このシャーシに収容されているシャーシである。
4. "Links"セクションで"ContainedBy"オブジェクトを調べる。
    1. "@odata.id"の値を探し出す。これは、このシャーシが収容されているシャーシである。

改訂履歴

バージョン	日付	説明
1.0.0	2015 年 8 月	初期リリース
1.0.1	2015 年 8 月	@DMTF を@Redfish に修正
1.0.2	2016 年 4 月	モックアップの Copyright 注釈に関する声明を追加