REST サービスの作成

Version 2023.1
2024-01-02

InterSystems Corporation　1 Memorial Drive　Cambridge MA 02142　www.intersystems.com

REST サービスの作成
InterSystems IRIS Data Platform　Version 2023.1　2024-01-02

InterSystems Worldwide Response Center (WRC)
Tel:　+1-617-621-0700
Tel:　+44 (0) 844 854 2917
Email:　support@InterSystems.com

目次

1

REST サービスの作成の概要

このページでは、REST の概要と InterSystems IRIS® における REST サービスの概要を紹介します。この REST インタフェースを UI ツール (Angular など) で使用すると、データベースや相互運用プロダクションにアクセスできます。また、これを使用して、外部システムから InterSystems IRIS® データ・プラットフォーム・アプリケーションにアクセスできるようにすることもできます。実際の操作を通じた REST サービスの説明は、"Developing REST Interfaces" を参照してください。

1.1 REST の概要

REST ("Representational State Transfer" の略称) には、以下の属性があります。

- REST とは、形式ではなく、アーキテクチャのスタイルです。REST は、多くの場合、メッセージの転送に HTTP を使用し、データの受け渡しに JSON を使用します。ただし、XML またはプレーン・テキストでデータを渡すこともできます。REST は、既存の Web 標準 (HTTP、URL、XML、JSON など) を利用します。
- REST は、リソース指向です。一般に、リソースは URL で識別され、明示的に HTTP メソッド (GET、POST、PUT、および DELETE など) に基づいた操作を使用します。
- 一般に、REST には小さいオーバーヘッドがあります。データの記述には XML を使用できますが、軽量のデータ・ラッパである JSON を使用する方が一般的です。JSON では、データをタグで識別しますが、そのタグは公式のスキーマ定義では指定されず、明示的なデータ型を持ちません。

1.2 インターシステムズの REST サービスの概要

InterSystems IRIS 2019.2 以降で REST インタフェースを定義するには、以下の 2 つの方法があります。

- 仕様優先の定義 – 最初に OpenAPI 2.0 仕様を作成してから、API 管理ツールを使用して REST インタフェースのコードを生成します。
- REST インタフェースの手動コーディング。

仕様優先の定義を使用すると、インターシステムズの REST サービスは形式的には以下のコンポーネントで構成されます。

- 仕様クラス (**%REST.Spec** のサブクラス)。このクラスには、REST サービスの OpenAPI 2.0 仕様が含まれます。インターシステムズでは、仕様内で使用できるさまざまな拡張属性をサポートしています。

- ディスパッチ・クラス (%CSP.REST のサブクラス)。このクラスは、HTTP 要求を受け取り、実装クラスの適切なメソッドを呼び出します。
- 実装クラス (%REST.Impl のサブクラス)。このクラスでは、REST 呼び出しを実装するメソッドを定義します。

  API 管理ツールによって、実装クラスのスタブ・バージョンが生成されます。その後、これを拡張して、必要なアプリケーション・ロジックを追加します (追加するロジックからこのクラスの外部のコードを呼び出すこともできます)。

  %REST.Impl クラスは、HTTP ヘッダの設定やエラーの報告などを行うために呼び出すことができるメソッドを提供します。
- インターシステムズの Web アプリケーション。インターシステムズの Web ゲートウェイを介して REST サービスへのアクセスを提供します。Web アプリケーションは、REST アクセスを有効にし、特定のディスパッチ・クラスを使用するように構成します。また、REST サービスへのアクセスを制御します。

インターシステムズでは、これらのコンポーネントについて厳格な名前付け規約に従います。アプリケーション名 (appname) を指定すると、仕様クラス、ディスパッチ・クラス、および実装クラスの名前はそれぞれ、appname.spec、appname.disp、および appname.impl になります。既定では、Web アプリケーションの名前は /csp/appname ですが、別の名前を使用することもできます。

インターシステムズでは、仕様優先のパラダイムをサポートしています。仕様から初期コードを生成でき、仕様が変更された場合は (新しいエンド・ポイントの取得による変更など)、そのコードを再生成できます。この後のセクションで詳しく説明しますが、この時点では、ディスパッチ・クラスを編集してはいけないことを覚えておいてください。ただし、他のクラスは変更可能です。また、仕様クラスをリコンパイルすると、ディスパッチ・クラスが自動的に再生成され、実装クラスが更新されます (編集内容は保持されます)。

1.2.1 REST サービスの手動コーディング

2019.2 より前のリリースの InterSystems IRIS では、仕様優先のパラダイムがサポートされていませんでした。REST サービスは、形式的にはディスパッチ・クラスと Web アプリケーションのみで構成されます。このドキュメントでは、この方法で定義する REST サービスを手動コーディングの REST サービスと呼びます。新しい REST サービスによって定義された REST サービスには仕様クラスが含まれるのに対して、手動コーディングの REST サービスには含まれない点が異なります。このドキュメントの付録 “手動による REST サービスの作成” では、手動コーディングのパラダイムを使用して REST サービスを作成する方法について説明します。同様に、いくつかの API 管理ユーティリティを使用して、手動コーディングの REST サービスを操作することもできます。

1.3 インターシステムズの API 管理ツールの概要

REST サービスをより簡単に作成できるように、インターシステムズでは以下の API 管理ツールを提供しています。

- /api/mgmnt という名前の REST サービス。このサービスを使用して、サーバ上の REST サービスを検出したり、それらの REST サービスの OpenAPI 2.0 仕様を生成したりすることができます。また、サーバで REST サービスの作成、更新、または削除を行うこともできます。
- ^%REST ルーチン。このルーチンは、REST サービスのリスト、作成、および削除に使用できるシンプルなコマンド行インタフェースを提供します。
- %REST.API クラス。このクラスを使用して、サーバ上の REST サービスを検出したり、それらの REST サービスの OpenAPI 2.0 仕様を生成したりすることができます。また、サーバで REST サービスの作成、更新、または削除を行うこともできます。

この章で後述する説明に従って、これらのツールのロギングを設定できます。

有用なサードパーティ・ツールには、PostMan (https://www.getpostman.com/) や Swagger エディタ (https://swagger.io/tools/swagger-editor/download/) などの REST テスト・ツールがあります。

1.4 REST サービスの作成の概要

インターシステムズ製品で REST サービスを作成するための大まかな方法は、以下のとおりです。この手順に従うことをお勧めします。

1. サービスの OpenAPI 2.0 仕様を入手 (または作成) します。
2. API 管理ツールを使用して、REST サービス・クラスおよび関連する Web アプリケーションを生成します。“REST サービスの作成と編集”を参照してください。
3. 適切なビジネス・ロジックがメソッドに含まれるように、実装クラスを変更します。“実装クラスの変更”の章を参照してください。
4. オプションで仕様クラスを変更します。“仕様クラスの変更”の章を参照してください。例えば、CORS のサポートや Web セッションの使用が必要な場合にこの手順を実行します。
5. セキュリティが必要な場合は、“REST サービスの保護”の章を参照してください。
6. サービスの OpenAPI 2.0 仕様を使用して、“REST API の検出およびドキュメント化”の章に記載されている説明に従ってドキュメントを生成します。

手順 2 については、仕様クラスを手動で作成し (仕様をクラスに貼り付ける)、そのクラスをコンパイルするという別のオプションもあります。このプロセスによって、ディスパッチ・クラスとスタブ実装クラスが生成されます。つまり、/api/mgmnt サービスまたは ^%REST ルーチンを必ず使用しなければならないというわけではありません。このドキュメントでは、この手法についてはこれ以上説明しません。

1.5 REST サービス・クラスの詳細

ここでは、仕様クラス、ディスパッチ・クラス、および実装クラスについて詳しく説明します。

1.5.1 仕様クラス

仕様クラスは、REST サービスが従うべきコントラクトを定義するためのものです。このクラスは %REST.Spec を拡張し、REST サービスの OpenAPI 2.0 仕様が含まれる XData ブロックを含みます。以下は、部分的な例です。

```
Class petstore.spec Extends %REST.Spec [ ProcedureBlock ]
{

XData OpenAPI [ MimeType = application/json ]
{
{
  "swagger":"2.0",
  "info":{
    "version":"1.0.0",
    "title":"Swagger Petstore",
    "description":"A sample API that uses a petstore as an example to demonstrate features in the
swagger-2.0 specification",
    "termsOfService":"http://swagger.io/terms/",
    "contact":{
      "name":"Swagger API Team"
    },
    "license":{
      "name":"MIT"
    }
  },
...
```

このクラスを変更するには、XData ブロック内の仕様を置換または編集します。クラスのパラメータ、プロパティ、およびメソッドを必要に応じて追加することもできます。仕様クラスをコンパイルすると、コンパイラによって必ずディスパッチ・クラスが再生成され、実装クラスが更新されます（“インターシステムズにおける実装クラスの更新の動作”を参照してください）。

1.5.2 ディスパッチ・クラス

ディスパッチ・クラスは、REST サービスが呼び出されたときに直接呼び出されます。以下は、部分的な例です。

```
/// Dispatch class defined by RESTSpec in petstore.spec
Class petstore.disp Extends %CSP.REST [ GeneratedBy = petstore.spec.cls, ProcedureBlock ]
{

/// The class containing the RESTSpec which generated this class
Parameter SpecificationClass = "petstore.spec";

/// Default the Content-Type for this application.
Parameter CONTENTTYPE = "application/json";

/// By default convert the input stream to Unicode
Parameter CONVERTINPUTSTREAM = 1;

/// The default response charset is utf-8
Parameter CHARSET = "utf-8";

XData UrlMap [ XMLNamespace = "http://www.intersystems.com/urlmap" ]
{
<Routes>
  <Route Url="/pets" Method="get" Call="findPets" />
  <Route Url="/pets" Method="post" Call="addPet" />
  <Route Url="/pets/:id" Method="get" Call="findPetById" />
  <Route Url="/pets/:id" Method="delete" Call="deletePet" />
</Routes>
}

/// Override %CSP.REST AccessCheck method
ClassMethod AccessCheck(Output pAuthorized As %Boolean) As %Status
{
   ...
}

...
```

SpecificationClass パラメータは、関連する仕様クラスの名前を示します。UrlMap XData ブロック（URL マップ）では、この REST サービス内の呼び出しを定義します。クラスのこの部分を詳しく理解する必要はありません。

これらの項目の後、URL マップにリストされているメソッドの定義がクラスに含まれます。以下に一例を示します。

```
ClassMethod deletePet(pid As %String) As %Status
{
    Try {
        If '##class(%REST.Impl).%CheckAccepts("application/json") Do
##class(%REST.Impl).%ReportRESTError(..#HTTP406NOTACCEPTABLE,$$$ERROR($$$RESTBadAccepts)) Quit
        If ($number(pid,"I")="") Do
##class(%REST.Impl).%ReportRESTError(..#HTTP400BADREQUEST,$$$ERROR($$$RESTInvalid,"id",id)) Quit
        Set response=##class(petstore.impl).deletePet(pid)
        Do ##class(petstore.impl).%WriteResponse(response)
    } Catch (ex) {
        Do ##class(%REST.Impl).%ReportRESTError(..#HTTP500INTERNALSERVERERROR,ex.AsStatus())
    }
    Quit $$$OK
}
```

以下の点に注意します。

- このメソッドは、実装クラス (この例では `petstore.impl`) 内で同じ名前でメソッドを呼び出します。これは、そのメソッドからの応答を取得し、%WriteResponse() を呼び出して応答を呼び出し元に書き込みます。%WriteResponse() メソッドは継承されたメソッドであり、すべての実装クラス (すべて %REST.Impl のサブクラス) に存在します。
- このメソッドは他のチェックを行い、エラーの場合は %REST.Impl の他のメソッドを呼び出します。

重要　ディスパッチ・クラスは生成されたクラスなので、絶対に編集しないでください。インターシステムズでは、ディスパッチ・クラスを編集せずに、その一部をオーバーライドするためのメカニズムを用意しています。

1.5.3 実装クラス

実装クラスは、REST サービスの実際の内部実装を格納するためのものです。このクラスは編集可能です (また、その必要があります)。最初は以下の例のような状態です。

```
/// A sample API that uses a petstore as an example to demonstrate features in the swagger-2.0
specification<br/>
/// Business logic class defined by RESTSpec in petstore.spec<br/>
Class petstore.impl Extends %REST.Impl [ ProcedureBlock ]
{

/// If ExposeServerExceptions is true, then details of internal errors will be exposed.
Parameter ExposeServerExceptions = 0;

/// Returns all pets from the system that the user has access to<br/>
/// The method arguments hold values for:<br/>
///     tags, tags to filter by<br/>
///     limit, maximum number of results to return<br/>
ClassMethod findPets(tags As %ListOfDataTypes(ELEMENTTYPE="%String"), limit As %Integer) As %Stream.Object
{
    //(Place business logic here)
    //Do ..%SetStatusCode(<HTTP_status_code>)
    //Do ..%SetHeader(<name>,<value>)
    //Quit (Place response here) ; response may be a string, stream or dynamic object
}

...
```

実装クラスの残りの部分には、これと同じような追加のスタブ・メソッドが含まれます。いずれの場合も、これらのスタブ・メソッドには、REST サービスの仕様で定義されたコントラクトに従ったシグニチャがあります。`options` メソッドについては、実装するスタブ・メソッドは生成されません。代わりに、クラス %CSP.REST によって、`options` のすべての処理が自動的に実行されます。

1.6 API 管理機能のロギングの有効化

API 管理機能のロギングを有効にするには、ターミナルで以下を入力します。

```
set $namespace="%SYS"
kill ^ISCLOG
set ^%ISCLOG=5
set ^%ISCLOG("Category","apimgmnt")=5
```

これによって、API 管理エンドポイントへのあらゆる呼び出しに使用する ^ISCLOG グローバルにエントリが追加されます。

ログをファイルに書き込むには (読みやすくするため)、以下を入力します (引き続き、%SYS ネームスペース内です)。

```
do ##class(%OAuth2.Utils).DisplayLog("filename")
```

filename は、作成するファイルの名前です。このディレクトリは既に存在している必要があります。ファイルが既に存在する場合は、そのファイルが上書きされます。

ログを停止するには、以下を入力します (引き続き、%SYS ネームスペース内です)。

```
set ^%ISCLOG=0
set ^%ISCLOG("Category","apimgmnt")=0
```

1.6.1 ログの表示

HTTP 要求のログが有効になると、ログ・エントリが ^ISCLOG グローバルに保存されます。このグローバルは %SYS ネームスペースにあります。

管理ポータルを使用してログを表示するには、**[システムエクスプローラ]→[グローバル]** に移動して、ISCLOG グローバル (%ISCLOG) ではないことに注意) を表示します。%SYS ネームスペースにいることを確認します。

2

REST サービスの作成と編集

InterSystems IRIS では、いくつかの方法で REST サービスを作成および変更できます。主な方法として、`/api/mgmnt/` サービスの呼び出し、^%REST ルーチンの使用、%REST.API クラスの使用の 3 つがあります。REST サービスを作成するこれら 3 種類の方法では、サービス・クラスの生成に使用する REST サービスの OpenAPI 2.0 (Swagger とも呼ばれます) 記述を作成する必要があります。サードパーティによって定義された REST サービスを実装する場合、そのサードパーティによってこの OpenAPI 2.0 の記述が提供されることがあります。OpenAPI 2.0 の記述の形式の詳細は、OpenAPI 2.0 仕様を参照してください。

サービス・クラスを生成した後、REST サービスの詳しい構築手順を "実装クラスの変更" および "仕様クラスの変更" で確認します。

2.1 /Api/mgmnt/ サービスの使用

REST サービスを作成、更新、および削除する方法の 1 つでは `/api/mgmnt/` サービスを呼び出します。

このサービスには、Web サービスのリストおよびドキュメント化に使用できるオプションも用意されています。

2.1.1 /api/mgmnt を使用した REST サービスの作成

2.1.1.1 /api/mgmnt/ を使用したサービス・クラスの生成

最初の手順では、REST サービス・クラスを以下のように生成します。

1. REST サービスの OpenAPI 2.0 の記述 (JSON 形式) を作成または入手します。
2. PostMan (https://www.getpostman.com/) などの REST テスト・ツールを入手します。
3. テスト・ツールで、HTTP 要求メッセージを以下のように作成します。
   - HTTP アクションとして、POST を選択または指定します。
   - URL として、以下の形式の URL を指定します。

     ```
     http://localhost:52773/api/mgmnt/v2/namespace/myapp
     ```

     localhost はサーバの名前です。52773 は、InterSystems IRIS が実行されている Web サーバ・ポートです。namespace は、REST サービスを作成するネームスペースです。myapp は、クラスを作成するパッケージの名前です。
   - 要求の本文として、Web サービスの OpenAPI 2.0 の記述 (JSON 形式) を貼り付けます。
   - 要求の本文のタイプを `JSON (application/json)` として指定します。

- IRISUsername パラメータと IRISPassword パラメータの値を指定します。IRISUsername については、%Developer ロールのメンバであり、指定されたネームスペースに対する読み取り/書き込みアクセス権を持っているユーザを指定します。

4. 要求メッセージを送信します。

   呼び出しが成功すると、InterSystems IRIS は、指定されたパッケージおよびネームスペースに **disp** クラス、**impl** クラス、および **spec** クラスを作成します。

5. テスト・ツールで、応答メッセージを確認します。要求が成功した場合、応答メッセージは以下の例のようになります。

```
{
    "msg": "New application myapp created"
}
```

基本の REST サービスを完成させるには、インターシステムズの Web アプリケーションを作成し、実装を定義します（“実装クラスの変更” の章を参照してください）。これらの手順を実行する順序はどちらが先でもかまいません。

2.1.1.2 Web アプリケーションの作成

この手順では、REST サービスへのアクセスを提供する Web アプリケーションを作成します。管理ポータルで、以下の手順を実行します。

1. **[システム管理]→[セキュリティ]→[アプリケーション]→[Web アプリケーション]** の順にクリックします。
2. **[新規 Web アプリケーション作成]** をクリックします。
3. 以下の値を指定します。
   - **名前** ― Web アプリケーションの名前。InterSystems IRIS のこのインスタンス内で一意でなければなりません。最も一般的なのは、Web アプリケーションが実行されるネームスペースに基づいた名前（`/csp/namespace`）です。
   - **ネームスペース** ― クラスを生成したネームスペースを選択します。
   - **アプリケーション有効** ― このチェック・ボックスにチェックを付けます。
   - **有効** ― [REST] を選択します。
   - **ディスパッチ・クラス** ― ディスパッチ・クラスの完全修飾名を入力します。これは常に `package.disp` でなければなりません。package は、生成されたクラスが格納されているパッケージの名前です。

   このページの他のオプションの詳細は、″アプリケーションの作成″ を参照してください。
4. **[保存]** をクリックします。

2.1.2 /api/mgmnt を使用した REST サービスの更新

インターシステムズの API 管理ツールを使用すると、実装クラスに加えた編集内容を変更することなく、生成されたクラスを更新することができます。実装クラスは必要に応じて再生成されますが、編集内容は保持されます。

更新を正常に完了すると、InterSystems IRIS によって、指定したパッケージに **disp** クラスと **spec** クラスが再生成され、編集内容を保持した状態で **impl** クラスが更新されます。応答メッセージは以下の例のようになります。

```
{
    "msg": "Application myapp updated"
}
```

2.1.2.1 インターシステムズにおける実装クラスの更新の動作

impl クラスを以前に編集した場合、それらの編集内容は以下のように保持されます。

- すべてのメソッドの実装は、そのまま残ります。
- 新しく追加したクラス・メンバは、そのまま残ります。

ただし、クラスおよび生成された各メソッドの説明 (/// コメント) は再生成されます。実装メソッドのシグニチャが変更された場合は (仕様が変更されたことによるものなど)、シグニチャが更新され、そのクラス・メソッドに以下のコメントが追加されます。

```
/// WARNING: This method's signature has changed.
```

2.1.3 /api/mgmnt を使用した REST サービスの削除

インターシステムズの API 管理ツールを使用すると、REST サービスを簡単に削除することもできます。そのためには、以下のように操作します。

1. REST テスト・ツールを使用して、HTTP 要求メッセージを以下のように作成します。
   - HTTP アクションとして、DELETE を選択または指定します。
   - URL として、以下の形式の URL を指定します。

     ```
     http://localhost:52773/api/mgmnt/v2/namespace/myapp
     ```

     localhost はサーバの名前です。52773 は、InterSystems IRIS が実行されている Web サーバ・ポートです。namespace は、REST サービスを作成するネームスペースです。myapp は、REST サービス・クラスが格納されているパッケージの名前です。
   - IRISUsername パラメータと IRISPassword パラメータの値を指定します。IRISUsername については、%Developer ロールのメンバであり、指定されたネームスペースに対する読み取り/書き込みアクセス権を持っているユーザを指定します。
2. 要求メッセージを送信します。

   呼び出しが成功すると、InterSystems IRIS は、指定されたパッケージおよびネームスペース内の **disp** クラスと **spec** クラスを削除します。

   ただし、**impl** クラスは削除しません。
3. テスト・ツールで、応答メッセージを確認します。要求が成功した場合、応答メッセージは以下の例のようになります。

   ```
   {
       "msg": "Application myapp deleted"
   }
   ```
4. 実装クラスを手動で削除します。

   安全のために、**/api/mgmnt** サービスでは、実装クラスが自動的に削除されることはありません。実装クラスには、非常に多くのカスタマイズが含まれている可能性があるからです。
5. この REST サービスについて以前に作成した Web アプリケーション (ある場合) を削除します。そのためには、以下のように操作します。
   a. 管理ポータルで、**[システム管理]→[セキュリティ]→[アプリケーション]→[Web アプリケーション]** の順にクリックします。
   b. Web アプリケーションが表示されている行で **[削除]** をクリックします。
   c. **[OK]** をクリックして削除を確定します。

2.2 ^%REST ルーチンの使用

^%REST ルーチンは、シンプルなコマンド行インタフェースです。いずれのプロンプトでも、以下の回答を入力できます。

^	前の質問に戻ります。
?	現在のオプションをすべてリストするメッセージが表示されます。
q または quit（大文字/小文字の区別なし）	ルーチンを終了します。

また、それぞれの質問では、その質問に対する既定の回答が括弧で囲んで表示されます。

2.2.1 ^%REST を使用したサービスの作成

REST サービスを作成する際には、REST サービスの OpenAPI 2.0 仕様から開始し、それを使用して REST サービス・クラスを生成することをお勧めします。^%REST ルーチンを使用してこの作業を行うには、以下の手順を実行します。

1. REST サービスの OpenAPI 2.0 仕様 (JSON 形式) を入手します。仕様をファイルとして保存するか、仕様にアクセスできる URL を書き留めます。
2. ターミナルで、REST サービスを定義するネームスペースに移動します。
3. 以下のコマンドを入力して、^%REST ルーチンを開始します。

   ```
   do ^%REST
   ```

4. 最初のプロンプトで、REST サービスの名前を入力します。この名前は、生成されたクラスのパッケージ名として使用されます。有効なパッケージ名を使用してください。`list`、`l`、`quit`、または `q` という名前を使用する場合は (大文字/小文字のどのような組み合わせでも)、名前を二重引用符で囲みます。例えば、`"list"` のように入力します。
5. 次のプロンプトで、Y (大文字/小文字の区別なし) と入力して、このサービスを作成することを確認します。

   続いて、使用する OpenAPI 2.0 仕様の場所を入力するように求められます。完全パス名または URL を入力します。
6. 次のプロンプトで、Y (大文字/小文字の区別なし) と入力して、この仕様を使用することを確認します。

   このネームスペースの指定したパッケージ内に disp クラス、impl クラス、および spec クラスが作成されます。その後、以下のような出力が表示されます。

   ```
   -----Creating REST application: myapp-----
   CREATE myapp.spec
   GENERATE myapp.disp
   CREATE myapp.impl
   REST application successfully created.
   ```

   次に、Web アプリケーションも作成するかどうかを尋ねられます。この Web アプリケーションを使用して、REST サービスにアクセスします。
7. この時点では、以下の操作を行うことができます。
   - Y (大文字/小文字の区別なし) と入力して、Web アプリケーションを今すぐ作成します。
   - N (大文字/小文字の区別なし) と入力して、ルーチンを終了します。

     “Web アプリケーションの作成” に記載されている説明に従って、Web アプリケーションを別途作成することができます。
8. Y と入力した場合、Web アプリケーションの名前を入力するように求められます。

この名前は、InterSystems IRIS のこのインスタンス内で一意でなければなりません。既定では、Web アプリケーションが実行されるネームスペースに基づいた名前 (/csp/namespace) です。

Web アプリケーションの名前を入力するか、Enter キーを押して既定の名前をそのまま使用します。

その後、以下のような出力が表示されます。

```
-----Deploying REST application: myapp-----
Application myapp deployed to /csp/myapp
```

9. “実装クラスの変更”の章に記載されている説明に従って、実装を定義します。

2.2.2 ^%REST を使用したサービスの削除

^%REST ルーチンを使用して REST サービスを削除するには、以下の手順を実行します。

1. ターミナルで、REST サービスがあるネームスペースに移動します。
2. 以下のコマンドを入力して、^%REST ルーチンを開始します。

```
do ^%REST
```

3. 最初のプロンプトで、REST サービスの名前を入力します。

   REST サービスの名前が不明な場合は、L (大文字/小文字の区別なし) と入力します。すべての REST サービスがリストされ、REST サービスの名前の入力を求めるプロンプトが再度表示されます。
4. 指定した名前の REST サービスが見つかると、以下のようなプロンプトが表示されます。

```
REST application found: petstore
Do you want to delete the application? Y or N (N):
```

5. Y (大文字/小文字の区別なし) と入力して、このサービスを削除することを確認します。
6. (オプション) 実装クラスを手動で削除します。

   安全のために、このルーチンでは、実装クラスが自動的に削除されることはありません。実装クラスには、非常に多くのカスタマイズが含まれている可能性があるからです。

2.3 %REST.API クラスの使用

このセクションでは、%REST.API クラスを使用して REST サービスを作成、更新、および削除する方法について説明します。

2.3.1 %REST.API クラスを使用したサービスの作成または更新

REST サービスを作成する際には、REST サービスの OpenAPI 2.0 仕様から開始し、それを使用して REST サービス・クラスを生成することをお勧めします。**%REST.API** クラスを使用してこの作業を行うには、以下の手順を実行します。

1. REST サービスの OpenAPI 2.0 仕様 (JSON 形式) を入手し、仕様をファイルとして保存します。

   このファイルは、UTF-8 でエンコードされている必要があります。
2. REST サービスを定義するネームスペースで、このファイルを使用して **%DynamicObject** のインスタンスを作成します。

3. 次に、%REST.API クラスの CreateApplication() メソッドを呼び出します。このメソッドには、以下のシグニチャがあります。

```
classmethod CreateApplication(applicationName As %String,
                              swagger As %DynamicObject = "",
                              ByRef features,
                              Output newApplication As %Boolean,
                              Output internalError As %Boolean)
                              as %Status
```

   引数は以下のとおりです。

   - applicationName は、クラスを生成するパッケージの名前です。
   - swagger は、OpenAPI 2.0 仕様を表す %DynamicObject のインスタンスです。

     この引数は、仕様の URL、仕様が格納されているファイルのパス名、または空の文字列として指定することもできます。
   - 参照渡しにする必要がある features は、追加オプションを保持する多次元配列です。
     - features("addPing") が 1 で、swagger が空の文字列である場合、生成されたクラスには、テストを目的とした ping() メソッドが含まれます。
     - features("strict") が 1 (既定値) である場合、仕様内のすべてのプロパティがチェックされます。features("strict") が 0 である場合は、コードの生成に必要なプロパティのみがチェックされます。
   - 出力として返される newApplication は、メソッドによって新しいアプリケーションが作成されたのか (true)、既存のアプリケーションが更新されたのかを示すブーリアン値です。
   - 出力として返される internalError は、内部エラーが発生したかどうかを示すブーリアン値です。

   メソッドによって新しいアプリケーションが生成された場合、InterSystems IRIS は、指定されたパッケージに disp クラス、impl クラス、および spec クラスを作成します。

   メソッドによって既存のアプリケーションが更新された場合、InterSystems IRIS は、指定されたパッケージ内の disp クラスと spec クラスを再生成し、編集内容を保持して impl クラスを更新します。

   OpenAPI 2.0 仕様が無効な場合、変更は行われません。

4. "REST サービスの作成と編集" の説明に従って、REST サービスにアクセスする Web アプリケーションを作成します。
5. "実装クラスの変更" の章に記載されている説明に従って、実装を定義します。

以下に最初の手順の例を示します。

```
set file="c:/2downloads/petstore.json"
set obj = ##class(%DynamicAbstractObject).%FromJSONFile(file)
do ##class(%REST.API).CreateApplication("petstore",.obj,,.new,.error)
//examine error and decide how to proceed...
...
```

2.3.2 %REST.API クラスを使用したサービスの削除

%REST.API クラスを使用して REST サービスを削除するには、以下の手順を実行します。

1. REST サービスがあるネームスペースで、%REST.API クラスの DeleteApplication() メソッドを呼び出します。このメソッドには、以下のシグニチャがあります。

```
classmethod DeleteApplication(applicationName As %String) as %Status
```

   applicationName は、REST サービス・クラスが格納されているパッケージの名前です。

2. (オプション) 実装クラスを手動で削除します。

   安全のために、このクラス・メソッドでは、実装クラスが自動的に削除されることはありません。実装クラスには、非常に多くのカスタマイズが含まれている可能性があるからです。

3. この REST サービスについて以前に作成した Web アプリケーション (ある場合) を削除します。そのためには、以下のように操作します。

   a. 管理ポータルで、**[システム管理]→[セキュリティ]→[アプリケーション]→[Web アプリケーション]** の順にクリックします。

   b. Web アプリケーションが表示されている行で **[削除]** をクリックします。

   c. **[OK]** をクリックして削除を確定します。

3
実装クラスの変更

この章では、REST サービスの実装クラスを変更する方法について説明します。

この章は、“REST サービスの作成と編集”の手順に従って REST サービス・クラスを生成済みであることを前提としています。

3.1 初期のメソッド定義

実装クラスには最初、以下の例のようなスタブ・メソッドが含まれています。

```
/// Returns all pets from the system that the user has access to<br/>
/// The method arguments hold values for:<br/>
///     tags, tags to filter by<br/>
///     limit, maximum number of results to return<br/>
ClassMethod findPets(tags As %ListOfDataTypes(ELEMENTTYPE="%String"), limit As %Integer) As %Stream.Object
{
    //(Place business logic here)
    //Do ..%SetStatusCode(<HTTP_status_code>)
    //Do ..%SetHeader(<name>,<value>)
    //Quit (Place response here) ; response may be a string, stream or dynamic object
}
```

いずれの場合も、これらのスタブ・メソッドには、REST サービスの仕様で定義されたコントラクトに従ったシグニチャがあります。

3.2 メソッドの実装

実装クラスの各メソッドについて、それを使用する REST 呼び出しに応じてメソッド定義 (具体的には実装) を編集します。メソッドの前に、対応する REST 呼び出しの説明のコピーであるコメントが含まれています。この実装では、以下の処理を行います。

- 適切な値を返します。
- 要求メッセージを検証します。そのためには、実装クラスの %CheckAccepts() メソッド、%GetContentType() メソッド、および %GetHeader() メソッドを使用します。ここで説明するメソッドはすべて、実装クラスのスーパークラスである **%REST.Impl** クラスから継承されます。
- 必要に応じて HTTP ステータス・コードを設定して、リソースが使用可能だったかどうかなどを示します。そのためには、%SetStatusCode() メソッドを使用します。HTTP ステータス・コードについては、http://www.faqs.org/rfcs/rfc2068.html を参照してください。

- HTTP 応答ヘッダを設定します。そのためには、%SetHeader() メソッド、%SetHeaderIfEmpty() メソッド、および %DeleteHeader() メソッドを使用します。
- 必要に応じてエラーを報告します。そのためには、%LogError() メソッドを使用します。

これらのメソッドの詳細は、**%REST.Impl** のクラスリファレンスを参照してください。

3.3 サーバ・エラーの詳細の公開

既定では、REST サービスで内部エラーが発生した場合、エラーの詳細はクライアントに報告されません。これを変更するには、実装クラスに以下のパラメータを追加した後、リコンパイルします。

```
Parameter ExposeServerExceptions = 1;
```

既定の %ReportRESTError() メソッドは、このパラメータをチェックします。そのメソッドをオーバーライドする場合 (次の見出しを参照)、メソッドでこのパラメータを使用するかどうかを選択できます。

3.4 エラー応答の変更

エラー応答の形式を既定以外の方法で設定する必要がある場合は、実装クラスの %ReportRESTError() メソッドをオーバーライドします。独自のメソッドで、%WriteResponse() メソッドを使用してエラー応答を返します。

これらのメソッドの詳細は、**%REST.Impl** のクラスリファレンスを参照してください。

4

仕様クラスの変更

この章では、REST サービスの仕様クラスを変更する方法とその理由を簡単に説明します。

この章は、“REST サービスの作成と編集”の手順に従って REST サービス・クラスを生成済みであることを前提としています。

4.1 概要

以下のテーブルは、仕様クラスを変更する理由をリストし、必要な変更内容を簡単にまとめたものです。

理由	変更内容
仕様を更新または置換する	手動で、または仕様クラスを再生成することにより、OpenAPI XData ブロックを変更します。
REST サービスで CORS のサポートを有効にする	手動で OpenAPI XData ブロックを変更します。また、クラス・パラメータを追加し、カスタム・ディスパッチ・スーパークラスを作成します。“REST サービスでの CORS のサポート”を参照してください。
REST サービスで Web セッションのサポートを有効にする	クラス・パラメータを追加します。“REST での Web セッションの使用”の章を参照してください。
エンドポイントを使用するために必要な特権を指定する	手動で OpenAPI XData ブロックを変更します。“REST サービスの保護”の章を参照してください。
既定のコンテンツ・タイプ、応答の文字セット、または入力ストリームの処理をオーバーライドする	クラス・パラメータを追加します。この章の次のセクションを参照してください。
サービス・メソッドに既定以外の名前を指定する	手動で OpenAPI XData ブロックを変更します。この章の“サービス・メソッドの名前のオーバーライド”を参照してください。

仕様クラスをコンパイルすると、コンパイラによって、同じパッケージにディスパッチ・クラスが必ず再生成され、実装クラスが更新されます（“インターシステムズにおける実装クラスの更新の動作”を参照してください）。

4.2 コンテンツ・タイプ、応答の文字セット、または入力ストリームの処理のオーバーライド

クラス・パラメータを仕様クラスに追加し、リコンパイルするだけで、REST サービスのいくつかの重要な要素をオーバーライドすることができます。

- 既定では、REST サービスは `application/json` のコンテンツ・タイプを想定します。これをオーバーライドするには、以下のパラメータを仕様クラスに追加します。

  ```
  Parameter CONTENTTYPE = "some-content-type";
  ```

  some-content-type は MIME コンテンツ・タイプです。

- 既定では、REST サービスの応答メッセージは UTF-8 形式です。これをオーバーライドするには、以下のパラメータを仕様クラスに追加します。

  ```
  Parameter CHARSET = "some-character-set";
  ```

  some-content-type-here は文字セットの名前です。

- 既定では、REST サービスは入力文字ストリームを Unicode に変換します。この処理を行わないようにするには、以下のパラメータを仕様クラスに追加します。

  ```
  Parameter CONVERTINPUTSTREAM = 0";
  ```

その後、リコンパイルします。これにより、これらの変更内容がディスパッチ・クラスにコピーされます。

4.3 サービス・メソッドの名前のオーバーライド

既定では、コンパイラは、操作の `operationId` を使用して、対応する REST 呼び出しによって呼び出されるメソッドの名前を決定します。別の名前を指定することもできます。そのためには、仕様クラスの `OpenAPI` XData ブロック内の操作に以下を追加します。

```
"x-ISC_ServiceMethod":"alternatename"
```

以下に例を示します。

```
"/pets":{
  "get":{
    "description":"Returns all pets from the system that the user has access to",
    "operationId":"findPets",
    "x-ISC_ServiceMethod":"ReturnPets",
    "produces":[
      "application/json",
      "application/xml",
      "text/xml",
      "text/html"
    ],
```

その後、リコンパイルします。これにより、この新しいメソッドがディスパッチ・クラスと実装クラスに追加されます。必ず、実装クラスを編集し、この新しいメソッドの実装を提供してください。

4.4 REST サービスでの CORS のサポート

CORS (Cross-Origin Resource Sharing) を使用すると、別のドメインで実行されているスクリプトからサービスにアクセスすることができます。

通常、ブラウザは 1 つのドメインからスクリプトを実行している場合、同じドメインへの XMLHttpRequest の呼び出しを許可しますが、別のドメインへ呼び出される場合は許可しません。このブラウザ動作により、機密データを不正使用するおそれのある悪意のあるスクリプトの作成が制限されます。悪意のあるスクリプトにより、ユーザは与えられた許可を使用して別のドメインの情報にアクセスできますが、ユーザの知らない者が機密情報を他の用途に用いることもできます。こうしたセキュリティの問題を回避するため、通常ブラウザはこの種のクロスドメイン呼び出しを許可しません。

CORS (Cross-Origin Resource Sharing) を使用しない場合、REST サービスにアクセスするスクリプトを使用する Web ページは通常、REST サービスを提供するサーバと同じドメインにある必要があります。一部の環境では、スクリプトを使用する Web ページを REST サービス提供サーバとは異なるドメインに置くと便利です。CORS はこのような配置を可能にします。

以下ではブラウザが CORS で XMLHttpRequest を処理する方法を簡略化して説明します。

1. ドメイン DomOne にある Web ページのスクリプトにはドメイン DomTwo にある InterSystems IRIS REST サービスへの XMLHttpRequest が含まれます。XMLHttpRequest には CORS のカスタム・ヘッダがあります。
2. ユーザはこの Web ページを閲覧してスクリプトを実行します。Web ページを含むドメインとは異なるドメインへの XMLHttpRequest をユーザのブラウザは検出します。
3. ユーザのブラウザは InterSystems IRIS REST サービスへ特別な要求を送信します。これは XMLHttpRequest の HTTP 要求メソッドおよび元の Web ページのドメイン (この例では DomOne) を示します。
4. 要求が許可されると、応答には要求された情報が含まれます。許可されない場合、CORS が要求を許可しなかったことを示すヘッダのみで応答は構成されます。

4.4.1 REST サービスでの CORS のサポートの有効化に関する概要

既定では、インターシステムズの REST サービスは CORS ヘッダを許可しません。ただし、CORS のサポートを有効にすることができます。REST サービスでの CORS のサポートの有効化には、以下の 2 つの部分があります。

- 一部またはすべての HTTP 要求について REST サービスが CORS ヘッダを受け入れるようにします。“CORS ヘッダの受け入れ”を参照してください。
- REST サービスが CORS 要求を検証し、処理を続行するかどうかを決定するコードを記述します。例えば、信頼されたスクリプトのみを含むドメインを含む許可リストを提供することができます。InterSystems IRIS には、ドキュメント用に単純な既定の実装が用意されています。この既定の実装では、すべての CORS 要求が許可されます。

重要　　既定の CORS ヘッダの処理は、機密データを扱う REST サービスには適していません。

4.4.2 CORS ヘッダの受け入れ

REST サービスが CORS ヘッダを受け入れるように指定するには、以下の手順を実行します。

1. 仕様クラスを変更して HandleCorsRequest パラメータを追加します。

   すべての呼び出しについて CORS ヘッダの処理を有効にするには、HandleCorsRequest パラメータを 1 として指定します。

   ```
   Parameter HandleCorsRequest = 1;
   ```

または、すべてではなく、一部の呼び出しについて CORS ヘッダの処理を有効にするには、HandleCorsRequest パラメータを "" (空の文字列) として指定します。

```
Parameter HandleCorsRequest = "";
```

2. HandleCorsRequest パラメータを "" として指定した場合、どの呼び出しで CORS をサポートするかを示すために、仕様クラスの OpenAPI XData ブロックを編集します。具体的には、操作オブジェクトについて、以下のプロパティ名および値を追加します。

   ```
   "x-ISC_CORS":true
   ```

   例えば、OpenAPI XData ブロックに以下のような内容が含まれているとします。

   ```
   "post":{
     "description":"Creates a new pet in the store.  Duplicates are allowed",
     "operationId":"addPet",
     "produces":[
       "application/json"
     ],
     ...
   ```

   x-ISC_CORS プロパティを以下のように追加します。

   ```
   "post":{
     "description":"Creates a new pet in the store.  Duplicates are allowed",
     "operationId":"addPet",
     "x-ISC_CORS":true,
     "produces":[
        "application/json"
     ],
     ...
   ```

3. 仕様クラスをコンパイルします。このアクションによってディスパッチ・クラスが再生成され、実際に動作が変更されます。ディスパッチ・クラスについて詳しく理解する必要はありませんが、以下のように変更されます。

   - HandleCorsRequest パラメータについて指定した値が含まれるようになります。
   - UrlMap XData ブロックに、変更した操作に対応する <Route> 要素について Cors="true" が含まれるようになります。

HandleCorsRequest パラメータが 0 (既定値) である場合、CORS ヘッダの処理はすべての呼び出しについて無効になります。この場合、CORS ヘッダを含む要求を REST サービスが受け取ると、サービスはその要求を拒否します。

重要　InterSystems IRIS REST サービスでは、OPTIONS 要求 (CORS プリフライト要求) がサポートされます。これは、REST サービスで CORS がサポートされているかどうかを判別するために使用します。そのような要求を送信するユーザには、REST サービスで使用するあらゆるデータベースに対する READ 許可が必要です。この許可がないと、サービスは HTTP 404 エラーで応答します。代行認証を使用する構成では、認証されたユーザが要求を送信します。ZAUTHENTICATE ルーチンでユーザに適切な許可を割り当てます。代行認証を使用しない構成では、この要求は認証なしで送信され、CSPSystem ユーザが実行します。管理ポータルを使用して適切な許可をユーザに割り当てます。

4.4.3 CORS ヘッダの処理方法の定義

REST サービスが CORS ヘッダを受け入れるようにした場合、既定では、サービスはすべての CORS 要求を受け入れます。REST サービスが CORS 要求を検証し、処理を続行するかどうかを決定するようにする必要があります。例えば、信頼されたスクリプトのみを含むドメインを含む許可リストを提供することができます。そのためには、以下の作業を行う必要があります。

- %CSP.REST のサブクラスを作成します。このクラスで、最初のサブセクションの説明に従って OnHandleCorsRequest() メソッドを実装します。

- 仕様クラスを変更してリコンパイルし、ディスパッチ・クラスを再生成します。

この結果、ディスパッチ・クラスは、%CSP.REST ではなく、カスタム・クラスを継承するため、既定の CORS ヘッダの処理をオーバーライドする OnHandleCorsRequest() の定義を使用するようになります。

4.4.3.1 OnHandleCorsRequest() の定義

%CSP.REST のサブクラスで、OnHandleCorsRequest() メソッドを定義します。このメソッドで CORS 要求を検証し、応答ヘッダを適宜設定する必要があります。

このメソッドを定義するには、CORS プロトコルの詳細に精通している必要があります (ここでは説明しません)。

また、要求を検証し、応答ヘッダを設定する方法を理解しておくことも必要です。そのためには、既定で使用されるメソッド、つまり %CSP.REST の HandleDefaultCorsRequest() メソッドを確認すると役立ちます。このセクションでは、このメソッドが起源、資格情報、ヘッダ、および要求メソッドをどのように処理するかを説明し、バリエーションを提示します。この情報を使用して、独自の OnHandleCorsRequest() メソッドを記述することができます。

以下のコードは起源を取得し、それを使用して応答ヘッダを設定します。バリエーションの候補の 1 つは、許可リストに照らして起源をテストすることです。ドメインが許可されている場合、応答ヘッダを設定します。許可されていない場合は、応答ヘッダを空の文字列に設定します。

```
#; Get the origin
Set tOrigin=$Get(%request.CgiEnvs("HTTP_ORIGIN"))

#; Allow requested origin
Do ..SetResponseHeaderIfEmpty("Access-Control-Allow-Origin",tOrigin)
```

以下の行では認証ヘッダが含まれるように指定します。

```
#; Set allow credentials to be true
Do ..SetResponseHeaderIfEmpty("Access-Control-Allow-Credentials","true")
```

以下の行では受信する要求からヘッダと要求メソッドを取得します。ヘッダと要求メソッドが許可されているかどうかをテストするコードを記述する必要があります。許可されている場合、それらを使用して応答ヘッダを設定します。許可されていない場合は、応答ヘッダを空の文字列に設定します。

```
#; Allow requested headers
Set tHeaders=$Get(%request.CgiEnvs("HTTP_ACCESS_CONTROL_REQUEST_HEADERS"))
Do ..SetResponseHeaderIfEmpty("Access-Control-Allow-Headers",tHeaders)

#; Allow requested method
Set tMethod=$Get(%request.CgiEnvs("HTTP_ACCESS_CONTROL_REQUEST_METHOD"))
Do ..SetResponseHeaderIfEmpty("Access-Control-Allow-Method",tMethod)
```

重要　　既定の CORS ヘッダの処理は、機密データを扱う REST サービスには適していません。

4.4.3.2 仕様クラスの変更

OnHandleCorsRequest() の実装を含む、%CSP.REST のカスタム・サブクラスを定義したら、以下の手順を実行します。

1. 新しいプロパティとして `x-ISC_DispatchParent` が `info` オブジェクトに追加されるように、仕様クラスの OpenAPI XData ブロックを編集します。このプロパティの値は、カスタム・クラスの完全修飾名でなければなりません。

例えば、OpenAPI XData ブロックが以下のようであるとします。

```
  "swagger":"2.0",
  "info":{
    "version":"1.0.0",
    "title":"Swagger Petstore",
    "description":"A sample API that uses a petstore as an example to demonstrate features in the
 swagger-2.0 specification",
    "termsOfService":"http://swagger.io/terms/",
    "contact":{
      "name":"Swagger API Team"
    },
...
```

%CSP.REST のカスタム・サブクラスの名前が `test.MyDispatchClass` であるとします。この場合、XData ブロックを以下のように変更します。

```
  "swagger":"2.0",
  "info":{
    "version":"1.0.0",
    "title":"Swagger Petstore",
    "description":"A sample API that uses a petstore as an example to demonstrate features in the
 swagger-2.0 specification",
    "termsOfService":"http://swagger.io/terms/",
    "x-ISC_DispatchParent":"test.MyDispatchClass",
    "contact":{
      "name":"Swagger API Team"
    },
...
```

2. 仕様クラスをコンパイルします。このアクションによって、ディスパッチ・クラスが再生成されます。このクラスは、カスタム・ディスパッチ・スーパークラスを拡張したものになります。そのため、独自に記述した OnHandleCorsRequest() メソッドを使用するようになります。

4.5 REST での Web セッションの使用

REST の目的の 1 つは、ステートレスにすることです。つまり、ある REST 呼び出しから次の呼び出しまでの間、サーバに情報を保存しないということです。複数の REST 呼び出しを通して 1 つの Web セッションを保持することは、ステートレス・パラダイムを崩すことになりますが、1 つの Web セッションを保持する理由が 2 つあります。

- 接続時間を最小限に抑える – REST 呼び出しごとに新しい Web セッションを作成する場合、REST 呼び出しによってサーバ上で新しいセッションを確立する必要があります。Web セッションを保持することで、REST 呼び出しの接続が高速化します。
- 複数の REST 呼び出し間でデータを保持する – 場合によっては、複数の REST 呼び出し間でデータを保持することが、ビジネス要件を効率的に満たすために必要になることがあります。

複数の REST 呼び出しにわたって 1 つの Web セッションを使用できるようにするには、仕様クラスで UseSession パラメータを 1 に設定します。以下に例を示します。

```
Parameter UseSession As Integer = 1;
```

その後、このクラスをリコンパイルします。

UseSession が 1 である場合、REST サービスの複数の呼び出しにわたって 1 つの Web セッションが保持されます。このパラメータが 0 (既定値) である場合は、REST サービスの呼び出しごとに新しい Web セッションが使用されます。

注釈　仕様クラスをリコンパイルすると、UseSession パラメータがディスパッチ・クラスにコピーされ、実際に動作が変更されます。

5
REST サービスの保護

REST サービスで機密データにアクセスする場合、サービスに認証を使用する必要があります。それぞれのユーザに異なるレベルのアクセス権を付与する必要がある場合は、エンドポイントに必要な特権も指定します。

この章は、“REST サービスの作成と編集”の手順に従って REST サービス・クラスを生成済みであることを前提としています。

5.1 REST サービスの認証の設定

InterSystems IRIS REST サービスでは、以下のいずれかの認証形式を使用できます。

- HTTP 認証ヘッダ – REST サービスにはこの認証形式を使用することをお勧めします。
- Web セッション認証 – URL で疑問符に続けてユーザ名とパスワードを指定します。
- OAuth 2.0 認証 – 次のサブセクションを参照してください。

5.1.1 REST アプリケーションおよび OAuth 2.0

OAuth 2.0 を使用して REST アプリケーションを認証するには、以下の作業をすべて行います。

- REST アプリケーションを含むリソース・サーバを、OAuth 2.0 リソース・サーバとして構成します。
- %Service.CSP の代行認証を許可します。
- 代行認証を使用するように Web アプリケーション (REST アプリケーション用) が構成されていることを確認します。
- %SYS ネームスペースで ZAUTHENTICATE という名前のルーチンを作成します。インターシステムズが提供するサンプル・ルーチン REST.ZAUTHENTICATE.mac をコピーして変更することができます。このルーチンは GitHub の Samples-Security サンプルに含まれています (https://github.com/intersystems/Samples-Security)。“InterSystems IRIS で使用するサンプルのダウンロード”で説明されているようにサンプル全体をダウンロードすることもできますが、単に GitHub でルーチンを開いて、その内容をコピーするほうが簡単です。

  ルーチンで、applicationName の値を変更し、必要に応じてその他の変更を加えます。

″OAuth 2.0 および OpenID Connect の使用法″ の “OAuth 2.0 クライアントとしての InterSystems IRIS Web アプリケーションの使用法”の章の“Web クライアントの代行認証の定義 (オプション)”も参照してください。

重要　HealthShare® で認証を使用している場合は、インターシステムズが提供する ZAUTHENTICATE ルーチンを使用する必要があります。独自のルーチンは作成できません。

5.2 REST サービスを使用するために必要な特権の指定

コードの実行やデータへのアクセスに必要な特権を指定するため、インターシステムズのテクノロジではロールベースのアクセス制御 (RBAC) を使用します。詳細は、"承認 : ユーザ・アクセスの制御" を参照してください。

それぞれのユーザに異なるレベルのアクセス権を付与する必要がある場合は、以下の作業を行って許可を指定します。

- 仕様クラスを変更して、REST サービス、または REST サービスの特定のエンドポイントを使用するために必要な特権を指定した後、リコンパイルします。特権は、許可 (読み取りや書き込みなど) をリソースの名前と組み合わせたものです。

  サブセクションを参照してください。

- 管理ポータルを使用して、以下の作業を行います。
  - 仕様クラスで参照するリソースを定義します。
  - 特権のセットを付与するロールを定義します。例えば、あるエンドポイントに対する読み取りアクセス権や別のエンドポイントに対する書き込みアクセス権をロールに付与できます。1 つのロールに特権のセットを複数含めることができます。
  - 各自のタスクに必要なすべてのロールにユーザを配置します。

さらに、%CSP.REST クラスの SECURITYRESOURCE パラメータを使用して承認を実行することもできます。

5.2.1 特権の指定

REST サービス全体について特権のリストを指定することも、エンドポイントごとに特権のリストを指定することもできます。そのためには、以下のように操作します。

1. サービスにアクセスするために必要な特権を指定するには、仕様クラス内の OpenAPI XData ブロックを編集します。`info` オブジェクトについて、`x-ISC_RequiredResource` という名前の新しいプロパティを追加します。このプロパティの値は、REST サービスのすべてのエンドポイントへのアクセスに必要となる、定義したリソースとそのアクセス・モード (resource:mode) のコンマ区切りリストです。

   以下に例を示します。

```
  "swagger":"2.0",
  "info":{
    "version":"1.0.0",
    "title":"Swagger Petstore",
    "description":"A sample API that uses a petstore as an example to demonstrate features in the
 swagger-2.0 specification",
    "termsOfService":"http://swagger.io/terms/",
    "x-ISC_RequiredResource":["resource1:read","resource2:read","resource3:read"],
    "contact":{
      "name":"Swagger API Team"
    },
...
```

2. 特定のエンドポイントにアクセスするために必要な特権を指定するには、以下の例のように、そのエンドポイントを定義する操作オブジェクトに `x-ISC_RequiredResource` プロパティを追加します。

```
      "post":{
        "description":"Creates a new pet in the store.  Duplicates are allowed",
        "operationId":"addPet",
        "x-ISC_RequiredResource":["resource1:read","resource2:read","resource3:read"],
        "produces":[
          "application/json"
        ],
        ...
```

3. 仕様クラスをコンパイルします。このアクションによって、ディスパッチ・クラスが再生成されます。

5.2.2 SECURITYRESOURCE パラメータの使用

追加の承認ツールとして、**%CSP.REST** のサブクラスであるディスパッチ・クラスには SECURITYRESOURCE パラメータがあります。SECURITYRESOURCE の値は、リソースとその許可であるか、リソースのみです (この場合、関連する許可は Use です)。SECURITYRESOURCE に関連付けられたリソースに対する必要な許可をユーザが持っているかどうかがチェックされます。

注釈 ディスパッチ・クラスで SECURITYRESOURCE の値を指定し、CSPSystem ユーザに十分な特権がない場合、ログイン試行の失敗に関する予期しない HTTP エラー・コードが返されることがあります。このようなエラーを回避するには、指定したリソースに対する許可を CSPSystem ユーザに付与することをお勧めします。

6

REST API のリストおよびドキュメント化

この章では、インスタンス上で使用可能な REST サービスを検出する方法と REST サービスのドキュメントを生成する方法について説明します。

6.1 /api/mgmnt サービスを使用した REST サービスの検出

/api/mgmnt サービスには、REST サービス・クラスおよび REST 対応 Web アプリケーションの検出に使用できる呼び出しが用意されています。

6.1.1 REST サービスの検出

/api/mgmnt サービスを使用して、インスタンス上で使用可能な REST サービスを検出するには、以下の REST 呼び出しを使用します。

- HTTP アクションとして、GET を選択または指定します。
- URL として、以下の形式の URL を指定します。

  ```
  http://localhost:52773/api/mgmnt/v2/
  ```

  または、1 つのネームスペースだけを調べる場合は、以下の形式の URL を指定します。

  ```
  http://localhost:52773/api/mgmnt/v2/:namespace
  ```

  localhost はサーバの名前、52773 は InterSystems IRIS が実行されている Web サーバ・ポート、namespace は、調べるネームスペースです。

(これらの呼び出しでは、手動コーディングの REST サービスは無視されます。手動コーディングの REST アプリケーションを検出するには、呼び出し GET /api/mgmnt/ および GET /api/mgmnt/:v1/:namespace/restapps を使用します。)

呼び出しが成功すると、InterSystems IRIS は、REST サービスをリストする JSON 形式の配列を返します。以下に例を示します。

```
[
  {
    "name": "%Api.Mgmnt.v2",
    "webApplications": "/api/mgmnt",
    "dispatchClass": "%Api.Mgmnt.v2.disp",
    "namespace": "%SYS",
    "swaggerSpec": "/api/mgmnt/v2/%25SYS/%Api.Mgmnt.v2"
  },
  {
    "name": "myapp",
    "webApplications": "/api/myapp",
    "dispatchClass": "myapp.disp",
    "namespace": "USER",
    "swaggerSpec": "/api/mgmnt/v2/USER/myapp"
  }
]
```

6.1.2 REST 対応 Web アプリケーションの検出

/api/mgmnt サービスを使用して、インスタンス上で使用可能な REST 対応 Web アプリケーションを検出するには、以下の REST 呼び出しを使用します。

- HTTP アクションとして、GET を選択または指定します。
- URL として、以下の形式の URL を指定します。

  ```
  http://localhost:52773/api/mgmnt
  ```

  または、1 つのネームスペースだけを調べる場合は、以下の形式の URL を指定します。

  ```
  http://localhost:52773/api/mgmnt/v1/:namespace/restapps
  ```

  localhost はサーバの名前、52773 は InterSystems IRIS が実行されている Web サーバ・ポート、namespace は、調べるネームスペースです。

"GET /api/mgmnt/" および "GET /api/mgmnt/:v1/:namespace/restapps" のリファレンス・セクションを参照してください。

6.2 %REST.API クラスを使用した REST サービスの検出

%REST.API クラスには、REST サービス・クラスおよび REST 対応 Web アプリケーションの検出に使用できるメソッドが用意されています。

6.2.1 REST サービス・クラスの検出

%REST.API クラスを使用して、インスタンス上で使用可能な REST サービスを検出するには、そのクラスの以下のメソッドを使用します。

GetAllRESTApps()

```
GetAllRESTApps(Output appList As %ListOfObjects) as %Status
```

このサーバ上の REST サービスのリストを出力として返します。出力引数 applist は、%ListOfObjects のインスタンスです。リスト内の各項目は、REST サービスに関する情報を含む %REST.Application のインスタンスです。これには、関連する Web アプリケーションがない REST サービスも含まれます。このメソッドでは、手動コーディングの REST サービスは無視されます。

GetRESTApps()

```
GetRESTApps(namespace as %String,
            Output appList As %ListOfObjects) as %Status
```

namepace で指定されたネームスペース内の REST サービスのリストを出力として返します。"GetAllWebRESTApps()" を参照してください。"GetAllRESTApps()" を参照してください。

6.2.2 REST 対応 Web アプリケーションの検出

%REST.API クラスを使用して、インスタンス上で使用可能な REST 対応 Web アプリケーションを検出するには、そのクラスの以下のメソッドを使用します。

GetAllWebRESTApps()

```
GetAllWebRESTApps(Output appList As %ListOfObjects) as %Status
```

このサーバ上の REST 対応 Web アプリケーションのリストを出力として返します。出力引数 applist は、%ListOfObjects のインスタンスです。リスト内の各項目は、Web アプリケーションに関する情報を含む %REST.Application のインスタンスです。

GetWebRESTApps()

```
GetWebRESTApps(namespace as %String,
               Output appList As %ListOfObjects) as %Status
```

namepace で指定されたネームスペース内の REST 対応 Web アプリケーションのリストを出力として返します。"GetAllWebRESTApps()" を参照してください。

6.3 REST サービスのドキュメントの提供

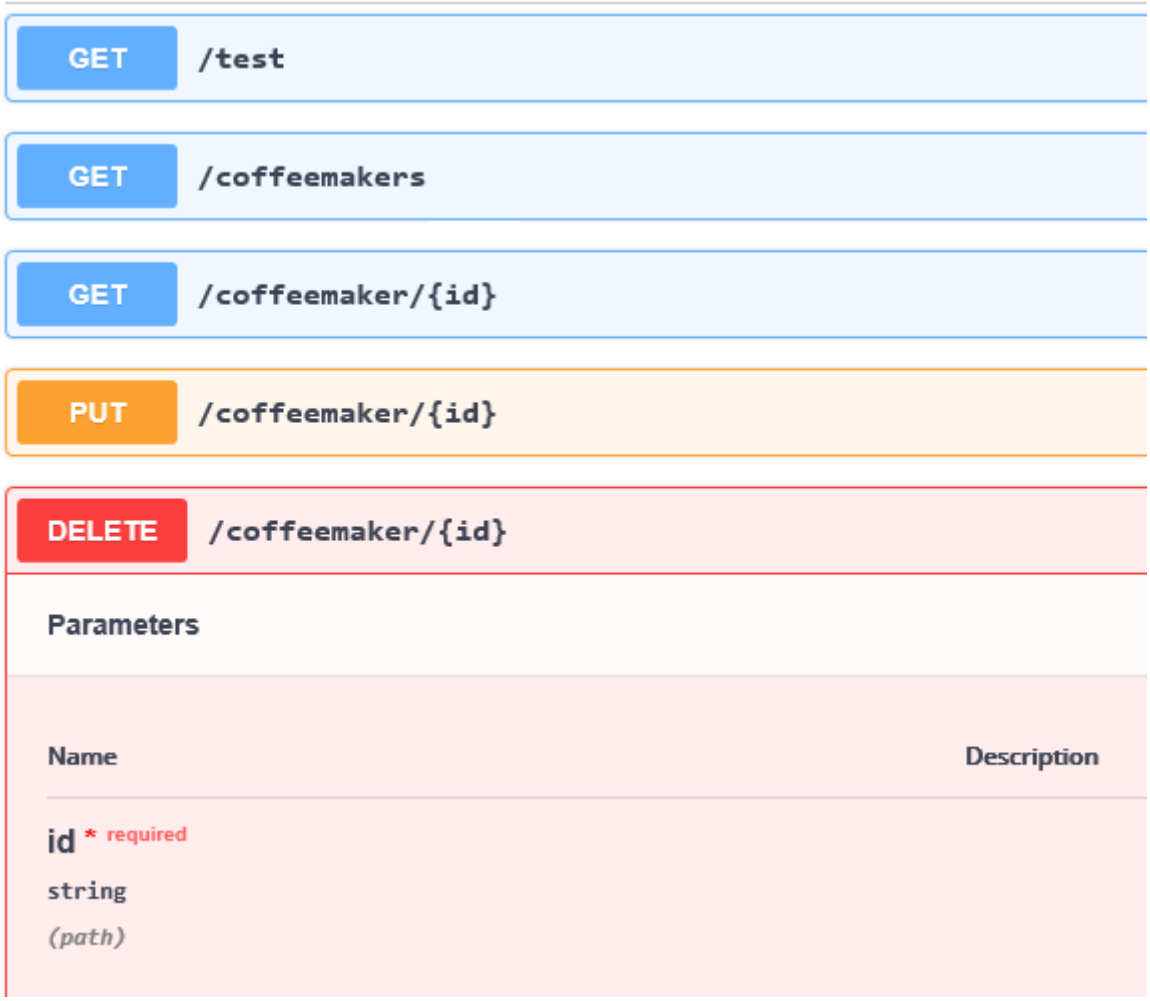
開発者が API を簡単に使用できるように、API をドキュメント化すると便利です。OpenAPI 2.0 仕様に従った REST API の場合、オープンソースの Swagger フレームワークを使用して、仕様の内容に基づいて API に関するインタラクティブなドキュメントを提供することができます。

1 つのオプションとして、Swagger UI を使用し、ドキュメントのホストされたコピーを提供する方法があります。デモを行うには、以下の手順を実行します。

1. https://swagger.io/tools/swagger-ui/ にアクセスします。
2. [Live Demo] をクリックします。
3. ページの上部にあるボックスに、REST サービスの OpenAPI 2.0 仕様 (JSON 形式) の URL を入力します。

   例えば、InterSystems IRIS サーバで GET /api/mgmnt/v2/:namespace/:application 呼び出しを使用します。
4. [Explore] をクリックします。

ページの下部に、以下の例のようなドキュメントが表示されます。



ここでは、それぞれの呼び出しの詳細を表示したり、テスト呼び出しを行って、応答を確認したりすることができます。詳細は、Swagger の Web サイトを参照してください。

他のサードパーティ・ツールを使用して、静的な HTML を生成することもできます。これについては特にお勧めするものはありません。

/api/mgmnt/ API エンドポイント

このリファレンスには、/api/mgmnt/ サービスのエンドポイントを掲載しています。これらはすべて、新しい REST サービスに適用されます。以下のテーブルに、各エンドポイントの概要とそれらが手動コーディングの REST サービスにも適用されるかどうかを示します。

エンドポイント	概要	新しい REST サービスに適用されるか	手動コーディングの REST サービスに適用されるか
DELETE /api/mgmnt/v2/:ns/:app	REST サービスを削除します。	はい	いいえ
GET /api/mgmnt/	このサーバ上の REST 対応 Web アプリケーションをリストします。	はい	はい
GET /api/mgmnt/v1/:ns/restapps	ネームスペース内の REST 対応 Web アプリケーションをリストします。	はい	はい
GET /api/mgmnt/v1/:ns/spec/:app	REST サービスの OpenAPI 2.0 仕様を返します。	いいえ	はい
GET /api/mgmnt/v2/	このサーバ上の REST サービス (関連する Web アプリケーションがないものも含む) をリストします。	はい	いいえ
GET /api/mgmnt/v2/:ns	ネームスペース内の REST サービス (関連する Web アプリケーションがないものも含む) をリストします。	はい	いいえ
GET /api/mgmnt/v2/:ns/:app	REST サービスの OpenAPI 2.0 仕様を返します。	はい	はい

ns はネームスペースです。app は、REST サービス・クラスが格納されているパッケージの名前です。

DELETE /api/mgmnt/v2/:namespace/:application/

指定された REST アプリケーションのクラスを削除します。この呼び出しは、新しい REST サービスを検索します。手動コーディングの REST サービスは無視します。

URL パラメータ

namespace	必須。ネームスペース名。このパラメータでは大文字と小文字は区別されません。
application	必須。spec クラス、impl クラス、および disp クラスが格納されているパッケージの完全修飾名。このパラメータでは大文字と小文字は区別されません。

許可

このエンドポイントを使用するには、%Developer ロールのメンバである必要があると共に、指定されたネームスペースに対する読み取り/書き込みアクセス権を持っている必要があります。

要求の例

- 要求のメソッド：

  ```
  DELETE
  ```

- 要求の URL：

  ```
  http://localhost:52773/api/mgmnt/v2/user/myapp
  ```

応答

この呼び出しに対する応答はありません。

GET /api/mgmnt/

すべてのネームスペース内の REST 対応 Web アプリケーションに関する情報を含む配列を返します。

URL パラメータ

なし。

許可

このエンドポイントを使用するには、指定されたネームスペースに対する読み取りアクセス権を持っている必要があります。ネームスペースを指定しない場合、または指定したネームスペースが %SYS である場合は、既定のネームスペース (USER) に対する読み取りアクセス権を持っている必要があります。既定のネームスペースを別のネームスペースに設定できます。そのためには、グローバル・ノード ^%SYS("REST","UserNamespace") を目的のネームスペースに設定します。

要求の例

- 要求のメソッド :

  ```
  GET
  ```

- 要求の URL :

  ```
  http://localhost:52773/api/mgmnt/
  ```

応答

応答は JSON 配列です。配列内の各オブジェクトは、このサーバ上の REST サービスを表します。具体的には、この呼び出しは、このサーバ上に構成されているすべての REST 対応 Web アプリケーションに関する情報を取得します。新しい REST サービスと手動コーディングの REST サービスの両方を検出します。関連する REST 対応 Web アプリケーションがない REST サービス・クラス (新しいものまたは手動コーディングのもの) がある場合、それらはこの応答に含まれません。

オブジェクトには以下のプロパティがあります。

- `name` ― REST 対応 Web アプリケーションの名前。
- `dispatchClass` ― REST サービスのディスパッチ・クラスの名前。具体的には、これは、Web アプリケーションの [**ディスパッチ・クラス**] 構成オプションで指定したクラスです。
- `namespace` ― ディスパッチ・クラスが定義されているネームスペース。
- `resource` ― この REST サービスを使用するために必要な InterSystems IRIS リソースの名前。
- `swaggerSpec` ― この REST サービスの OpenAPI 2.0 仕様を取得できるエンドポイント。
- `enabled` ― REST サービスが有効であるかどうかを指定します。

応答の例を以下に示します。

```
[
    {
        "name": "/api/atelier",
        "dispatchClass": "%Api.Atelier",
        "namespace": "%SYS",
        "resource": "%Development",
        "swaggerSpec": "/api/mgmnt/v1/%25SYS/spec/api/atelier",
        "enabled": true
    },
    {
        "name": "/api/deepsee",
```

```
        "dispatchClass": "%Api.DeepSee",
        "namespace": "%SYS",
        "resource": "",
        "swaggerSpec": "/api/mgmnt/v1/%25SYS/spec/api/deepsee",
        "enabled": true
    },
    {
        "name": "/api/docdb",
        "dispatchClass": "%Api.DocDB",
        "namespace": "%SYS",
        "resource": "",
        "swaggerSpec": "/api/mgmnt/v1/%25SYS/spec/api/docdb",
        "enabled": true
    },
    {
        "name": "/api/iknow",
        "dispatchClass": "%Api.iKnow",
        "namespace": "%SYS",
        "resource": "",
        "swaggerSpec": "/api/mgmnt/v1/%25SYS/spec/api/iknow",
        "enabled": true
    },
    {
        "name": "/api/mgmnt",
        "dispatchClass": "%Api.Mgmnt.v2.disp",
        "namespace": "%SYS",
        "resource": "",
        "swaggerSpec": "/api/mgmnt/v1/%25SYS/spec/api/mgmnt",
        "enabled": true
    },
    {
        "name": "/api/uima",
        "dispatchClass": "%Api.UIMA",
        "namespace": "%SYS",
        "resource": "",
        "swaggerSpec": "/api/mgmnt/v1/%25SYS/spec/api/uima",
        "enabled": true
    },
    {
        "name": "/webapp/simple2",
        "dispatchClass": "simple2.disp",
        "namespace": "USER",
        "resource": "",
        "swaggerSpec": "/api/mgmnt/v1/USER/spec/webapp/simple2",
        "enabled": true
    }
]
```


関連項目

- GET /api/mgmnt/v2/

GET /api/mgmnt/v1/:namespace/restapps

指定されたネームスペース内の REST 対応 Web アプリケーションに関する情報を含む配列を返します。

URL パラメータ

なし。

許可

このエンドポイントを使用するには、指定されたネームスペースに対する読み取りアクセス権を持っている必要があります。ネームスペースを指定しない場合、または指定したネームスペースが %SYS である場合は、既定のネームスペース (USER) に対する読み取りアクセス権を持っている必要があります。既定のネームスペースを別のネームスペースに設定できます。そのためには、グローバル・ノード ^%SYS("REST","UserNamespace") を目的のネームスペースに設定します。

要求の例

- 要求のメソッド :

  ```
  GET
  ```

- 要求の URL :

  ```
  http://localhost:52773/api/mgmnt/v1/user/restapps
  ```

応答

応答は JSON 配列です。配列内の各オブジェクトは REST 対応 Web アプリケーションを表します。詳細は、"GET /api/mgmnt/" を参照してください。

関連項目

- GET /api/mgmnt/v2/:namespace

GET /api/mgmnt/v1/:namespace/spec/:application/

指定された REST サービスの OpenAPI 2.0 仕様を返します。このサービスは、手動コーディングの REST サービスである必要があります。

URL パラメータ

namespace	必須。ネームスペース名。このパラメータでは大文字と小文字は区別されません。
application	必須。REST サービス・クラスが格納されているパッケージの完全修飾名。

許可

このエンドポイントを使用するには、指定されたネームスペースに対する読み取りアクセス権を持っている必要があります。ネームスペースを指定しない場合、または指定したネームスペースが %SYS である場合は、既定のネームスペース (USER) に対する読み取りアクセス権を持っている必要があります。既定のネームスペースを別のネームスペースに設定できます。そのためには、グローバル・ノード ^%SYS("REST","UserNamespace") を目的のネームスペースに設定します。

また、ディスパッチ・クラスが格納されているデータベースに対する読み取りアクセス権も必要です。ディスパッチ・クラスのネームスペースとエンドポイントのネームスペースは同じである必要があります (パッケージ内の % で始まるディスパッチ・クラスの場合を除きます。これはすべてのネームスペースに使用できます)。

要求の例

- 要求のメソッド :

  ```
  GET
  ```

- 要求の URL :

  ```
  http://localhost:52773/api/mgmnt/v1/user/spec/myapp
  ```

応答

この呼び出しは、https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md に記載されているような Swagger (OpenAPI 2.0) 仕様を返します。

GET /api/mgmnt/v2/

サーバ上の新しい REST サービス (関連する Web アプリケーションがないものも含む) に関する情報を含む配列を返します。この呼び出しは、手動コーディングの REST サービスは無視します。

URL パラメータ

なし。

許可

このエンドポイントを使用するには、指定されたネームスペースに対する読み取りアクセス権を持っている必要があります。ネームスペースを指定しない場合、または指定したネームスペースが %SYS である場合は、既定のネームスペース (USER) に対する読み取りアクセス権を持っている必要があります。既定のネームスペースを別のネームスペースに設定できます。そのためには、グローバル・ノード ^%SYS("REST","UserNamespace") を目的のネームスペースに設定します。

要求の例

- 要求のメソッド :

  ```
  GET
  ```

- 要求の URL :

  ```
  http://localhost:52773/api/mgmnt/v2/
  ```

応答

応答は JSON 配列です。配列内の各オブジェクトは REST サービスを表します。オブジェクトには以下のプロパティがあります。

- name — REST サービスの名前。
- webApplications — REST サービスへのアクセスを提供する Web アプリケーションの名前。
- dispatchClass — REST サービスのディスパッチ・クラスの名前。
- namespace — ディスパッチ・クラスおよびその他のクラスが定義されているネームスペース。
- swaggerSpec — この REST サービスの OpenAPI 2.0 仕様を取得できるエンドポイント。

応答の例を以下に示します。

```
[
  {
    "name": "%Api.Mgmnt.v2",
    "webApplications": "/api/mgmnt",
    "dispatchClass": "%Api.Mgmnt.v2.disp",
    "namespace": "%SYS",
    "swaggerSpec": "/api/mgmnt/v2/%25SYS/%Api.Mgmnt.v2"
  },
  {
    "name": "myapp",
    "webApplications": "/api/myapp",
    "dispatchClass": "myapp.disp",
    "namespace": "USER",
    "swaggerSpec": "/api/mgmnt/v2/USER/myapp"
  }
]
```

GET /api/mgmnt/v2/:namespace/

指定されたネームスペース内の新しい REST サービス (関連する Web アプリケーションがない REST サービスも含む) に関する情報を含む配列を返します。この呼び出しは、手動コーディングの REST サービスは無視します。

URL パラメータ

namespace	必須。ネームスペース名。このパラメータでは大文字と小文字は区別されません。

許可

このエンドポイントを使用するには、指定されたネームスペースに対する読み取りアクセス権を持っている必要があります。ネームスペースを指定しない場合、または指定したネームスペースが %SYS である場合は、既定のネームスペース (USER) に対する読み取りアクセス権を持っている必要があります。既定のネームスペースを別のネームスペースに設定できます。そのためには、グローバル・ノード ^%SYS("REST","UserNamespace") を目的のネームスペースに設定します。

要求の例

- 要求のメソッド :

  ```
  GET
  ```

- 要求の URL :

  ```
  http://localhost:52773/api/mgmnt/v2/%25sys/
  ```

応答

応答は JSON 配列です。配列内の各オブジェクトは REST サービスを表します。詳細は、“GET /api/mgmnt/v2/” を参照してください。

応答の例を以下に示します。

```
[
  {
    "name": "%Api.Mgmnt.v2",
    "webApplications": "/api/mgmnt",
    "dispatchClass": "%Api.Mgmnt.v2.disp",
    "namespace": "%SYS",
    "swaggerSpec": "/api/mgmnt/v2/%25SYS/%Api.Mgmnt.v2"
  }
]
```

GET /api/mgmnt/v2/:namespace/:application/

指定された REST サービスの OpenAPI 2.0 仕様を返します。この REST サービスは、新しい REST サービスでも手動コーディングの REST サービスでもかまいません。

URL パラメータ

namespace	必須。ネームスペース名。このパラメータでは大文字と小文字は区別されません。
application	必須。spec クラス、impl クラス、および disp クラスが格納されているパッケージの完全修飾名。このパラメータでは大文字と小文字は区別されません。

許可

このエンドポイントを使用するには、指定されたネームスペースに対する読み取りアクセス権を持っている必要があります。ネームスペースを指定しない場合、または指定したネームスペースが %SYS である場合は、既定のネームスペース (USER) に対する読み取りアクセス権を持っている必要があります。既定のネームスペースを別のネームスペースに設定できます。そのためには、グローバル・ノード ^%SYS("REST","UserNamespace") を目的のネームスペースに設定します。

また、ディスパッチ・クラスが格納されているデータベースに対する読み取りアクセス権も必要です。ディスパッチ・クラスのネームスペースとエンドポイントのネームスペースは同じである必要があります (パッケージ内の % で始まるディスパッチ・クラスの場合を除きます。これはすべてのネームスペースに使用できます)。

要求の例

- 要求のメソッド :

  ```
  GET
  ```

- 要求の URL :

  ```
  http://localhost:52773/api/mgmnt/v2/user/myapp
  ```

応答

この呼び出しは、https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md に記載されているような Swagger (OpenAPI 2.0) 仕様を返します。返される仕様では、プロパティが以下のように設定されます。

- `host` は常に、呼び出したサーバの server:port に設定されます。
- `basePath` は、この REST アプリケーションにディスパッチされる最初の Web アプリケーションに設定されます。この REST アプリケーションにディスパッチされる Web アプリケーションがない場合、`basePath` は、.spec クラスで定義されたとおりに返されます。

POST /api/mgmnt/v2/:namespace/:application

Swagger (OpenAPI 2.0) 仕様を指定すると、この呼び出しは REST アプリケーションのスキャフォールディングを生成します。

URL パラメータ

nam/espace	必須。ネームスペース名。このパラメータでは大文字と小文字は区別されません。
application	必須。spec クラス、impl クラス、および disp クラスが格納されているパッケージの完全修飾名。このパラメータでは大文字と小文字は区別されません。

要求の本文

要求の本文は、JSON 形式の Swagger (OpenAPI 2.0) 仕様である必要があります。

許可

このエンドポイントを使用するには、%Developer ロールのメンバである必要があると共に、指定されたネームスペースに対する読み取り/書き込みアクセス権を持っている必要があります。

要求の例

- 要求のメソッド :

  POST

- 要求の URL :

  ```
  http://localhost:52773/api/mgmnt/v2/user/myapp
  ```

- 要求の本文 :

  JSON 形式の Swagger (OpenAPI 2.0) 仕様。

A

使用される OpenAPI のプロパティ

この付録には、REST サービス・クラスを生成するときに API 管理ツールで使用される OpenAPI 2.0 仕様のプロパティを掲載しています。ここに掲載されていないプロパティは無視されます。拡張プロパティがいくつかあります。拡張プロパティの名前は `x-ISC` で始まります。

A.1 Swagger

- `basePath`
- `consumes`
- `host`
- `produces`
- `definitions` (API 管理ツールでは、コードを生成する際に Schema オブジェクトのいずれのプロパティも使用されません)
- `parameters` (詳細は、"Parameter オブジェクト" を参照してください)
- `paths` (詳細は、"Path Item オブジェクト" を参照してください)
- `info` (詳細は、"Info オブジェクト" を参照してください)
- `swagger` (`"2.0"` でなければなりません)

これらのプロパティの詳細は、https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md#swagger-object を参照してください。

A.2 Info オブジェクト

- `title`
- `description`
- `x-ISC_RequiredResource` (REST サービスのすべてのエンドポイントへのアクセスに必要となる、定義したリソースとそのアクセス・モード (resource:mode) のコンマ区切りリスト)
- `version`

標準プロパティの詳細は、https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md#info-object を参照してください。

A.3 Path Item オブジェクト

- `$ref`
- `get`、`put` など (OpenAPI 2.0 仕様に掲載されているすべてのメソッドがサポートされています)

  `options` メソッドについては、実装するスタブ・メソッドは生成されません。代わりに、クラス %CSP.REST によって、`options` のすべての処理が自動的に実行されます。
- `parameters` (詳細は、"Parameter オブジェクト" を参照してください)

これらのプロパティの詳細は、https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md#pathItemObject を参照してください。

A.4 Operation オブジェクト

- `operationId`
- `summary`
- `description`
- `consumes`
- `produces`
- `parameters` (詳細は、"Parameter オブジェクト" を参照してください)
- `x-ISC_CORS` (このエンドポイント/メソッドの組み合わせの CORS 要求をサポートするかどうかを示すフラグ)
- `x-ISC_RequiredResource` (REST サービスのこのエンドポイントへのアクセスに必要となる、定義したリソースとそのアクセス・モード (resource:mode) のコンマ区切りリスト)
- `x-ISC_ServiceMethod` (この操作を処理するためにバックエンドで呼び出されるクラス・メソッドの名前。既定値は `operationId` で、通常はこのままで適切です)
- `responses` (応答オブジェクト内では、`status` は HTTP ステータス・コードまたは `"default"` です)

標準プロパティの詳細は、https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md#operationObject を参照してください。

A.5 Parameter オブジェクト

- `name`
- `in`
- `description`

- required
- $ref
- type（"formData" にすることはできません。他のタイプは使用できます）
- format
- allowEmptyValue
- maxLength
- minLength
- pattern
- maximum
- minimum
- exclusiveMaximum
- exclusiveMinimum
- multipleOf
- collectionFormat
- minItems
- maxItems
- uniqueItems
- items（詳細は、“Items オブジェクト”を参照してください）

これらのプロパティの詳細は、https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md#parameter-object を参照してください。

A.6 Items オブジェクト

- type
- format
- allowEmptyValue
- maxLength
- minLength
- pattern
- maximum
- minimum
- exclusiveMaximum
- exclusiveMinimum
- multipleOf

- `collectionFormat`
- `minItems`
- `maxItems`
- `uniqueItems`

これらのプロパティの詳細は、https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md#items-object を参照してください。

B

手動による REST サービスの作成

この付録では、**%CSP.REST** クラスのサブクラスを作成することによって、InterSystems IRIS® REST サービスを手動で作成する方法について説明します。この手順で作成した手動コーディングの REST サービスは、すべての API 管理ツールで機能するとは限りません。

“REST サービスの保護”の“REST サービスの認証の設定”も参照してください。

B.1 手動による REST サービスの作成の基本

REST サービスを手動で定義するには、以下の手順を実行します。

- **%CSP.REST** のサブクラスである REST サービス・クラスを作成します。サブクラスで以下を実行します。
  - REST URL と HTTP メソッドに対して実行される InterSystems IRIS のメソッドを指定する URL マップを定義します。
  - オプションで、UseSession パラメータを指定します。このパラメータは、各 REST 呼び出しを独自の Web セッションで実行するか、1 つのセッションを他の REST 呼び出しと共有するかを制御します。
  - 必要な場合は、エラー処理のメソッドをオーバーライドできます。

  実装コードをディスパッチ・コードから分離する場合、REST サービスを実装するメソッドを別のクラスで定義し、それらのメソッドを URL マップから呼び出すことができます。
- REST サービス・クラスをディスパッチ・クラスとして使用する Web アプリケーションを定義します。

  Web アプリケーションとそのセキュリティを定義するには、**[Web アプリケーション]** ページに移動します (**[システム管理]→[セキュリティ]→[アプリケーション]→[Web アプリケーション]** の順にクリックします)。

  Web アプリケーションを定義する際には、**[ディスパッチ・クラス]** を、作成した REST サービス・クラスの名前に設定します。

  また、アプリケーションの名前を REST 呼び出しの URL の最初の部分として指定します。名前の例としては `/csp/mynamespace` や `/csp/myapp` がありますが、URL で使用できる任意のテキストを指定できます。

1 つのネームスペース内に REST サービス・クラスを複数定義できます。独自のエントリ・ポイントを有する REST サービス・クラスにはそれぞれ、独自の Web アプリケーションが必要です。

B.2 URL マップの作成

REST サービス・クラスで、サービスを実装するメソッドに REST 呼び出しを関連付ける、UrlMap という名前の XData ブロックを定義します。URL のコンテンツに基づいて呼び出しをメソッドに直接送信することも、URL に基づいて呼び出しを別の REST サービス・クラスに転送することもできます。Web アプリケーションが処理している関連するサービスが少数の場合、実装メソッドに呼び出しを直接送信することができます。一方、Web アプリケーションが多数の異なるサービスを処理する場合は別個の REST サービス・クラスを定義し、そのそれぞれで関連する一連のサービスを処理できます。その後、REST 呼び出しを他の REST サービス・クラスに適宜転送する中央の REST サービス・クラスを使用するように Web アプリケーションを構成します。

%CSP.REST のサブクラスが呼び出しをメソッドに直接送信する場合、UrlMap には、一連の <Route> 要素を含む <Routes> 定義が含まれます。それぞれの <Route> 要素で、指定した URL と HTTP 操作について呼び出すクラス・メソッドを指定します。一般的に、REST は GET、POST、PUT、または DELETE 操作を使用しますが、任意の HTTP 操作を指定することができます。URL には必要に応じてパラメータを含めることができます。このパラメータは、REST URL の一部として指定され、指定のメソッドにパラメータとして渡されます。

%CSP.REST のサブクラスが %CSP.REST の他のサブクラスに呼び出しを転送する場合、UrlMap には、一連の <Map> 要素を含む <Routes> 定義が含まれます。<Map> 要素は、指定された接頭語を持つすべての呼び出しを別の REST サービス・クラスに転送します。その後、そのクラスによって動作が実装されます。呼び出しをメソッドに直接送信する、または別のサブクラスに転送することで動作を実装できます。

重要　InterSystems IRIS は、受信した REST URL を各 <Route> の URL プロパティおよび各 <Map> の Prefix プロパティと比較します。その際、URL マップの最初の項目から始めて、同じ HTTP 要求メソッドを使用する最初に一致した項目で停止します。そのため、<Routes> 内の要素の順序は重要です。受信した URL が URL マップの複数の要素と一致する可能性がある場合、InterSystems IRIS は、最初に一致した要素を使用し、一致する可能性があるそれ以降の要素は無視します。

B.2.1 〈Route〉 要素を使用した UrlMap

InterSystems IRIS は、受信した URL および HTTP 要求メソッドを URL マップの各 <Route> 要素と比較します。そして、最初に一致した <Route> 要素で指定されているメソッドを呼び出します。<Route> 要素は、以下の 3 つの部分で構成されます。

- Url ― REST サービスを呼び出す REST URL の末尾部分の書式を指定します。Url は : (コロン) で始まるテキスト要素とパラメータから構成されます。
- Method ― REST 呼び出しの HTTP 要求メソッドを指定します。一般的には GET、POST、PUT、または DELETE ですが、任意の HTTP 要求メソッドを使用できます。要求メソッドには、サービスで実行される機能に適したものを選択する必要がありますが、%CSP.REST クラスはそれぞれのメソッドに関する特別な処理を実行しません。HTTP 要求メソッドはすべて大文字で指定する必要があります。
- Call ― REST サービスを実行するために呼び出すクラス・メソッドを指定します。既定では、このクラス・メソッドは REST サービス・クラスで定義されますが、任意のクラス・メソッドを明示的に指定することもできます。

例えば、以下の <Route> を考えてみましょう。

```
<Route Url="/echo" Method="POST" Call="Echo" Cors="false" />
```

これは、REST 呼び出しの最後が /echo になることと、POST メソッドを使用することを指定しています。これにより、REST サービスを定義する REST.DocServer クラスの Echo クラス・メソッドを呼び出します。Cors プロパティはオプションです。詳細は、“CORS を使用するための REST サービスの変更”を参照してください。

完全な REST URL は、以下の部分で構成されます。

- InterSystems IRIS サーバのサーバ名とポート。この章の例では、サーバ名およびポート `http://localhost:52773/` を使用しています。
- [Web アプリケーション] ページ ([システム管理]→[セキュリティ]→[アプリケーション]→[Web アプリケーション] の順にクリック) で定義された Web アプリケーションの名前(例 : `/csp/samples/docserver`)。
- `<Route>` 要素の `Url` プロパティ。`Url` プロパティのセグメントの先頭が : (コロン) の場合は、そのセグメントがパラメータであることを意味します。パラメータはその URL セグメントのどの値とも一致します。この値は、パラメータとしてメソッドに渡されます。

前述の例の場合、TCP トレース・ユーティリティで以下のように完全な REST 呼び出しが示されます。

```
POST /csp/samples/docserver/echo HTTP/1.1
Host: localhost:52773
```

REST サービスを実装するコードを %CSP.REST ディスパッチ・コードと区別する場合、REST サービスを実装するメソッドを別のクラスで定義し、Call 要素内でクラスとメソッドを指定できます。

B.2.1.1 パラメータの指定

以下の `<Route>` 定義では、URL 内に 2 つのパラメータ (namespace と class) が定義されています。

```
<Route Url="/class/:namespace/:classname" Method="GET" Call="GetClass" />
```

REST 呼び出しの URL は `/csp/samples/docserver/class/` で始まり、それに続く 2 つの URL の要素では 2 つのパラメータを指定します。GetClass() メソッドは、クエリするネームスペースおよびクラス名としてこれらのパラメータを使用します。例えば、以下の REST 呼び出しを考えてみましょう。

```
http://localhost:52773/csp/samples/docserver/class/samples/Cinema.Review
```

この REST 呼び出しは GetClass() メソッドを呼び出し、文字列 `"samples"` および `"Cinema.Review"` をパラメータ値として渡します。GetClass() メソッドには、以下のシグニチャがあります。

```
/// This method returns the class text for the named class
ClassMethod GetClass(pNamespace As %String,
                     pClassname As %String) As %Status
{
```

B.2.1.2 1 つの URL に対する複数のルートの指定

指定された URL について、異なる HTTP 要求メソッドをサポートできます。そのためには、HTTP 要求ごとに別個の ObjectScript メソッドを定義するか、要求を検証する 1 つの ObjectScript メソッドを使用します。

以下の例では、1 つの URL について HTTP 要求メソッドごとに異なるメソッドを使用しています。

```
<Route Url="/request" Method="GET" Call="GetRequest" />
<Route Url="/request" Method="POST" Call="PostRequest" />
```

このようなルーティングにより、HTTP GET メソッドで URL `/csp/samples/docserver/request` を呼び出したときには、GetRequest() メソッドが起動されます。HTTP POST メソッドで呼び出すと、PostRequest() メソッドが起動されます。

一方、以下の `<Route>` 定義を使用することもできます。

```
<Route Url="/request" Method="GET" Call="Request" />
<Route Url="/request" Method="POST" Call="Request" />
```

この場合は、Request() メソッドにより、GET 操作または POST 操作の呼び出しが処理されます。このメソッドにより、%CSP.Request のインスタンスである %request オブジェクトが検証されます。このオブジェクトでは、URL プロパティに URL のテキストが含まれています。

B.2.1.3 ルート・マップでの正規表現

ルート・マップ内で正規表現を使用できます。これは、ニーズを満たすために REST サービスを定義する方法が他にない場合にのみ行うことをお勧めします。このセクションで詳細に説明します (ObjectScript での正規表現の詳細は、"ObjectScript の使用法" の "正規表現" を参照してください)。

内部的には、URL 内でのパラメータの定義に使用する :parameter-name 構文は、正規表現を使用して実装されます。:parameter-name として指定された各セグメントは、繰り返す一致グループが含まれる正規表現、具体的には `([^/]+)` 正規表現に変換されます。この構文は、文字列 (長さがゼロ以外) に / (スラッシュ) 文字が含まれない限り、あらゆる文字列と一致します。したがって、GetClass() サンプル `Url="/class/:namespace/:classname"` は、以下と同等です。

```
<Route Url="/class/([^/]+)/([^/]+)" Method="GET" Call="GetClass" />
```

ここでは、2 つのパラメータを指定する 2 つの一致グループがあります。

ほとんどの場合、この形式で REST URL の指定に十分な柔軟性を提供しますが、上級ユーザはルート定義に正規表現形式を直接使用することができます。この URL は、正規表現およびそれぞれの一致グループに合致している必要があります。これは、1 組の括弧によって指定され、メソッドに渡されるパラメータを定義します。

例えば、以下のルート・マップを考えてみます。

```
<Routes>
<Route Url="/Move/:direction" Method="GET" Call="Move" />
<Route Url="/Move2/(east|west|north|south)" Method="GET" Call="Move" />
</Routes>
```

最初のルートでは、パラメータに任意の値を指定できます。パラメータの値にかかわらず、Move() メソッドが呼び出されます。2 つ目のルートでは、パラメータの値は `east`、`west`、`north`、`south` のいずれかである必要があります。これら以外の値を指定してこの 2 つ目のルートを呼び出した場合、Move() メソッドは呼び出されず、REST サービスから 404 エラーが返されます。これは、リソースが見つからないからです。

この簡単な例は、通常のパラメータ構文と正規表現との違いを示すことだけを目的としています。この例の場合、正規表現は必要ありません。Move() メソッドがパラメータの値を確認して適切に応答できるからです (またその必要があるからです)。ただし、以下の場合は、正規表現が役立ちます。

- パラメータがオプションの場合。この場合、:parameter-name 構文の代わりに、正規表現 `([^/]*)` を使用します。次に、例を示します。

  ```
  <Route Url="/Test3/([^/]*)" Method="GET" Call="Test"/>
  ```

  もちろん、ここで呼び出されるメソッドもパラメータに NULL 値が含まれる場合はこれを処理できる必要があります。

- パラメータが最後のパラメータで、値にスラッシュが含まれる場合。この場合、そのパラメータが必須であれば、:parameter-name 構文の代わりに正規表現 `((?s).+)` を使用します。次に、例を示します。

  ```
  <Route Url="/Test4/((?s).+)" Method="GET" Call="Test"/>
  ```

  または、そのパラメータがオプションであれば、:parameter-name 構文の代わりに正規表現 `((?s).*)` を使用します。次に、例を示します。

  ```
  <Route Url="/Test5/((?s).*)" Method="GET" Call="Test"/>
  ```

B.2.2 〈Map〉 要素を使用した UrlMap

InterSystems IRIS は、受信した URL を URL マップの各 <Map> 要素の接頭語と比較します。そして、最初に一致した <Map> 要素で指定されている REST サービス・クラスに、受信した REST 呼び出しを転送します。そのクラスは URL の残りの部分を処理し、サービスを実装するメソッドを通常呼び出します。<Map> 要素には以下の 2 つの属性があります。

- Prefix ― 照合する URL のセグメントを指定します。受信 URL には通常、一致するセグメントの後に他のセグメントがあります。
- Forward ― 一致するセグメントに続く URL セグメントを処理する別の REST サービス・クラスを指定します。

3 つの <Map> 要素を含む以下の UrlMap を考えてみましょう。

```
XData UrlMap
{
  <Routes>
    <Map Prefix="/coffee/sales" Forward="MyLib.coffee.SalesREST"/>
    <Map Prefix="/coffee/repairs" Forward="MyLib.coffee.RepairsREST"/>
    <Map Prefix="/coffee" Forward="MyLib.coffee.MiscREST"/>
  </Routes>
}
```

この UrlMap は、3 つの REST サービス・クラス (MyLib.coffee.SalesREST、MyLib.coffee.RepairsREST、または MyLib.coffee.MiscREST) のいずれかに REST 呼び出しを転送します。

これらの REST サービスの 1 つを呼び出す完全な REST URL は、以下の部分で構成されます。

- InterSystems IRIS サーバのサーバ名とポート (例えば、http://localhost:52773/)。
- [Web アプリケーション] ページ ([システム管理]→[セキュリティ]→[アプリケーション]→[Web アプリケーション] の順にクリック) で定義された Web アプリケーションの名前。例えば、これらの REST 呼び出しの Web アプリケーションの名前としては /coffeeRESTSvr などが考えられます。
- <Map> 要素の Prefix。
- REST URL の残りの部分。これは、転送された REST 要求を受け取る REST サービス・クラスによって処理される URL です。

例えば、以下の REST 呼び出しがあるとします。

```
http://localhost:52773/coffeeRESTSvr/coffee/sales/reports/id/875
```

これは、/coffee/sales という接頭語を持つ 1 つ目の <Map> と一致し、REST 呼び出しを MyLib.coffee.SalesREST クラスに転送します。そのクラスは URL の残りの部分 "/reports/id/875" で一致するものを探します。

別の例として、以下の REST 呼び出しがあるとします。

```
http://localhost:52773/coffeeRESTSvr/coffee/inventory/machinetype/drip
```

これは、/coffee という接頭語を持つ 3 つ目の <Map> と一致し、REST 呼び出しを MyLib.coffee.MiscREST クラスに転送します。そのクラスは URL の残りの部分 "/inventory/machinetype/drip" で一致するものを探します。

注釈　この UrlMap の例で、prefix="/coffee" の <Map> が 1 つ目のマップであった場合、/coffee を含む REST 呼び出しはすべて、それ以降の <Map> 要素のいずれかと一致した場合でも、MyLib.coffee.MiscREST クラスに転送されます。<Routes> 内の <Map> 要素の順序は重要です。

B.3 データ形式の指定

REST サービスは、さまざまな形式のデータ (JSON、XML、テキスト、CSV など) を処理するように定義できます。REST 呼び出しでは、HTTP 要求に ContentType 要素を指定することで、送信するデータに期待するフォームを指定できます。また、HTTP 要求に Accept 要素を指定することで返されるデータの形式を要求できます。

DocServer サンプルでは、GetNamespaces() メソッドによって、REST 呼び出しが JSON データを要求したかどうかが以下のようにチェックされます。

```
If $Get(%request.CgiEnvs("HTTP_ACCEPT"))="application/json"
```

B.4 REST サービスのローカライズ

REST サービスから返される文字列はすべてローカライズできるため、サーバはさまざまな言語で複数バージョンの文字列を格納できます。HTTP `Accept-Language` ヘッダが含まれる HTTP 要求をサービスが受け取ると、サービスは適切なバージョンの文字列で応答します。

REST サービスをローカライズするには、以下の操作を行います。

1. 実装コード内で、ハードコードされたリテラル文字列を含めるのではなく、`$$$Text` マクロのインスタンスを使用し、マクロ引数の値を以下のように指定します。
   - 既定の文字列
   - (オプション) この文字列の所属先のドメイン (文字列がドメインにグループ化されている場合、ローカライズの管理が簡単になります)
   - (オプション) 既定の文字列の言語コード

   例えば、以下を指定するのではなく、

   ```
   set returnvalue="Hello world"
   ```

   以下を指定します。

   ```
   set returnvalue=$$$TEXT("Hello world","sampledomain","en-us")
   ```

2. `$$$Text` マクロに対するドメイン引数を省略する場合は、REST サービス・クラス内に DOMAIN クラス・パラメータも指定します。次に、例を示します。

   ```
   Parameter DOMAIN = "sampledomain"
   ```

3. コードをコンパイルします。このとき、コンパイラによって、`$$$Text` マクロの一意のインスタンスごとに、メッセージ・ディクショナリにエントリが生成されます。

   メッセージ・ディクショナリはグローバルであるため、(例えば) 管理ポータルで簡単に確認できます。一般的なタスクに役立つクラス・メソッドがいくつかあります。

4. 開発が完了したら、対象のドメインまたはすべてのドメインのメッセージ・ディクショナリをエクスポートします。

   結果は、元の言語でのテキスト文字列を含む 1 つ以上の XML メッセージ・ファイルになります。

5. これらのファイルをトランスレータに送信し、変換されたバージョンを要求します。

6. 変換された XML メッセージ・ファイルを受け取ったら、元のファイルのエクスポート元である同じネームスペースにインポートします。

メッセージ・ディクショナリに、変換されたテキストと元のテキストが共存します。

7. 実行時に REST サービスは、HTTP `Accept-Language` ヘッダに基づいて、返すテキストを選択します。

詳細は、技術文書 "文字列のローカライズとメッセージ・ディクショナリ" を参照してください。

B.5 REST での Web セッションの使用

概要については、このドキュメントで前述した "REST での Web セッションの使用" を参照してください。

REST サービスで複数の REST 呼び出しにわたって 1 つの Web セッションを使用できるようにするには、REST サービス・クラスで UseSession パラメータを 1 に設定します。

```
Parameter UseSession As Integer = 1;
```

B.6 CORS のサポート

概要については、このドキュメントで前述した "CORS の概要" を参照してください。この付録の説明に従って Web サービスを手動で作成する場合、CORS のサポートの詳細は多少異なります。

B.6.1 CORS を使用するための REST サービスの変更

REST サービスで CORS をサポートするように指定するには、REST サービス・クラスを以下のように変更した後、リコンパイルします。

1. HandleCorsRequest パラメータの値を指定します。

   すべての呼び出しについて CORS ヘッダの処理を有効にするには、HandleCorsRequest パラメータを 1 として指定します。

   ```
   Parameter HandleCorsRequest = 1;
   ```

   または、すべてではなく、一部の呼び出しについて CORS ヘッダの処理を有効にするには、HandleCorsRequest パラメータを "" (空の文字列) として指定します。

   ```
   Parameter HandleCorsRequest = "";
   ```

   (HandleCorsRequest が 0 である場合、CORS ヘッダの処理はすべての呼び出しについて無効になります。この場合、CORS ヘッダを含む要求を REST サービスが受け取ると、サービスはその要求を拒否します。これが既定です。)

2. HandleCorsRequest パラメータを "" として指定した場合、どの呼び出しで CORS をサポートするのかを示すために、UrlMap XData ブロックを編集します。具体的には、CORS をサポートする必要がある `<Route>` について、以下のプロパティ名および値を追加します。

   ```
   Cors="true"
   ```

   または、`<Route>` 要素で `Cors="false"` を指定して、CORS の処理を無効にします。

REST サービス・クラスが別の REST サービス・クラスに REST 要求を転送する場合、CORS の処理の動作は、指定された要求と一致する `<Route>` 要素を含むクラスによって決まります。

B.6.2 CORS ヘッダの処理のオーバーライド

重要　　既定の CORS ヘッダの処理は、機密データを扱う REST サービスには適していません。

既定の CORS ヘッダの処理ではフィルタ処理は実行されず、CORS ヘッダが外部サーバに渡され、応答が返されるだけです。ドメインの許可リスト内の起源へのアクセスを制限するか、許可される要求メソッドを制限することをお勧めします。そのためには、REST サービス・クラスの OnHandleCorsRequest() メソッドをオーバーライドします。

OnHandleCorsRequest() メソッドの実装の詳細は、このドキュメントで前述した“OnHandleCorsRequest() の定義”を参照してください。

UrlMap 内の <Route> 要素と一致する URL 要求はすべて、クラス内で定義された 1 つの OnHandleCorsRequest() メソッドで処理されます。異なる REST URL 要求に対して OnHandleCorsRequest() メソッドの異なる実装が必要な場合は、Forward を使用して他の REST サービス・クラスに要求を送信する必要があります。

B.7 バリエーション：クエリ・パラメータへのアクセス

パラメータを REST サービスに渡す際には、サービスを呼び出すために使用される URL パスの一部として渡すことをお勧めします（例：/myapi/someresource/parametervalue）。ただし、パラメータをクエリ・パラメータとして渡す方が便利な場合もあります（例：/myapi/someresource?parameter=value）。そのような場合は、%request 変数を使用してパラメータ値を取得することができます。REST サービス内で、%request 変数は、URL クエリ全体を保持する %CSP.Request のインスタンスです。指定されたクエリ・パラメータの値を取得するには、以下の構文を使用します。

```
$GET(%request.Data(name,1),default)
```

name は、クエリ・パラメータの名前です。default は、返す既定値です。また、同じ URL が同じクエリ・パラメータの複数のコピーを保持する場合は、以下の構文を使用します。

```
$GET(%request.Data(name,index),default)
```

index は、取得するコピーの数値インデックスです。詳細は、%CSP.REST のクラスリファレンスを参照してください。

B.8 例：Hello World!

以下のコードは、きわめて簡単な REST サービスのサンプルです。helloWorld.disp、helloWorld.impl、helloWorld.hwObj の 3 つのクラスがあります。helloWorld.disp および helloWorld.impl は %CSP.REST を拡張して REST サービスを確立しますが、helloWorld.hwobj は %Persistent と %JSON.Adaptor の両方を拡張します。POST メソッドを使用してオブジェクトを作成するときに、この動作が便利です。

```
Class helloWorld.disp Extends %CSP.REST
{

Parameter HandleCorsRequest = 0;

XData UrlMap [ XMLNamespace = "https://www.intersystems.com/urlmap" ]
{
<Routes>
    <Route Url="/hello" Method="GET" Call="Hello" />
    <Route Url="/hello" Method="POST" Call="PostHello" />
</Routes>
}

ClassMethod Hello() As %Status
{
    Try {
```

```
        Do ##class(%REST.Impl).%SetContentType("application/json")
        If '##class(%REST.Impl).%CheckAccepts("application/json") Do
##class(%REST.Impl).%ReportRESTError(..#HTTP406NOTACCEPTABLE,$$$ERROR($$$RESTBadAccepts)) Quit
        Set response=##class(helloWorld.impl).Hello()
        Do ##class(%REST.Impl).%WriteResponse(response)
    } Catch (ex) {
        Do ##class(%REST.Impl).%SetStatusCode("400")
        return {"errormessage": "Client error"}
    }
    Quit $$$OK
}

ClassMethod PostHello() As %Status
{
    Try {
        Do ##class(%REST.Impl).%SetContentType("application/json")
        If '##class(%REST.Impl).%CheckAccepts("application/json") Do
##class(%REST.Impl).%ReportRESTError(..#HTTP406NOTACCEPTABLE,$$$ERROR($$$RESTBadAccepts)) Quit
        Set response=##class(helloWorld.impl).PostHello()
        Do ##class(%REST.Impl).%WriteResponse(response)
    } Catch (ex) {
        Do ##class(%REST.Impl).%SetStatusCode("400")
        return {"errormessage": "Client error"}
    }
    Quit $$$OK
}
```

上記のディスパッチ・クラスの構造に注意してください。さまざまなメソッドでさまざまなエンドポイントに URLMap が定義されていて、これらのルートを実装クラスにディスパッチするクラス・メソッドがあります。その過程で、このクラスによってエラー処理が実行されます。エラーの報告およびステータス・コードの設定の詳細は、%REST.Impl のドキュメントを参照してください。POST メソッドまたは PUT メソッドのエンドポイントであるクラス・メソッドは、特殊な %request パラメータを使用して、そのパラメータから実装メソッドに情報を渡す必要があります。

```
Class helloWorld.impl Extends %CSP.REST
{

ClassMethod Hello() As %DynamicObject
{
    Try {
       return {"Hello":"World"}.%ToJSON()
    } Catch (ex) {
        Do ##class(%REST.Impl).%SetStatusCode("500")
        return {"errormessage": "Server error"}
   }
}

ClassMethod PostHello(body As %DynamicObject) As %DynamicObject
{
    Try {
        set temp = ##class(helloWorld.hwobj).%New()
        Do temp.%JSONImport(body)
        Do temp.%Save()
        Do temp.%JSONExportToString(.ret)
        return ret
    } Catch (ex) {
        Do ##class(%REST.Impl).%SetStatusCode("500")
        return {"errormessage": "Server error"}
   }
}

}
```

上記のサンプルの実装クラスには 2 つのメソッドがあります。1 つは、JSON 形式の "Hello World" メッセージを返すだけのメソッドです。もう 1 つは、入力内容に基づいてオブジェクトを作成し、そのオブジェクトのコンテンツを返すメソッドです。%New() と %JSONImport() のようなメソッドの使用法の詳細は、それぞれ %Persistent および %JSON.Adapter の各ドキュメントを参照してください。

helloWorld.hwobj クラスには多数のプロパティを置くことができますが、簡略化のために、この例では 1 つのみとしています。

```
Class helloWorld.hwobj Extends (%Persistent, %JSON.Adaptor)
{

Property Hello As %String;

Storage Default
```

```
{
<Data name="hwobjDefaultData">
<Value name="1">
<Value>%%CLASSNAME</Value>
</Value>
<Value name="2">
<Value>Hello</Value>
</Value>
</Data>
<DataLocation>^helloWorld.hwobjD</DataLocation>
<DefaultData>hwobjDefaultData</DefaultData>
<IdLocation>^helloWorld.hwobjD</IdLocation>
<IndexLocation>^helloWorld.hwobjI</IndexLocation>
<StreamLocation>^helloWorld.hwobjS</StreamLocation>
<Type>%Storage.Persistent</Type>
}

}
```

"手動による REST サービスの作成の基本" に記載されている手順に従って管理ポータルで Web アプリケーションを構成した後、Postman などの REST クライアントを使用してその Web アプリケーションに要求を送信し、応答を確認します。