# API versioning

<http://www.troyhunt.com/2014/02/your-api-versioning-is-wrong-which-is.html>

As outlined in this excellent blog post there are three ways to version your rest api (plus model versioning)

My recommendation is the Custom request header.

**Why?**

With versioning and introducing versioning, we attempt to achieve the following goals:

* Be able to run multiple versions of a service which are live at the same time and ensure that the correct version of the service answers a request
* Be able to test for backwards compatibility
* Be able to test deployed services with integration tests regardless of the environment
* Avoid having to support legacy calls with the same code that is used for current calls
* Avoid having to update clients when a non-breaking change is released
* Avoid manually maintaining versioning
* Support non versioned API calls from legacy clients

Most importantly we want to do all that with minimum of fuss and maximising automated enforcement. I outline below a way of achieving this using the Request Header. You could do similar things using the other methods but essentially the Request Header is the least invasive way of attaching some meta data to the call. It requires minimal changes to code and once it’s in you can forget about it. Also importantly it doesn’t break any calls which don’t use it or break any services which aren’t expecting it.

What’s wrong with the other methods?

**Url includes version number**:

This forces clients to update their endpoints when a new version is released, even if the change is nonbreaking. Even in the case of breaking changes where the client would be updated in any case, I have found that it is difficult to manage the endpoint url change.

With externally facing services, decommissioning urls is hard.

Graceful fall back is hard if you are using virtual directories

The apparent upside of this method is you can implement the version routing in the code, so you don’t need a clever load balancer, or you can deploy versions to iis virtual directories.

However these are somewhat ‘cheap’ solutions and not ideal.

**Accept headers** :

I’ve not seen this done. It seems essentially the same as Customer headers, except it possibly causes bugs if you set it wrong.

**Model versioning:**

Not mentioned in the article, but another common method is to add a version field to each data model. I have found this onerous to support, with many manually set model version numbers. It doesn’t cover the case where you update a service without changing the models and is difficult to implement routing based on the model version in the content of the message.

However, this may well be a good idea for models which interact with the xls such as base variables?

By strongly linking the Service and Client to the model version via Specific references we ensure that we can determine the model version from the client version if required.

Code

**Models**

**Models** contains your Data Models which are consumed and produced by your service. It also optionally contains the interface for the service **IService** depending on how coupled they are. If not, put the Interface in its own project, which is also shared via nuget.

**Service**

**Service** is a class lib which implements **IService** and the business logic. This solution also contains a hosting project for the service. The service should expose an extra **ServiceInfo** method which returns the version number, status and configuration data for the service.

**ServiceRest** (the ‘client’)

**ServiceRest** which also implements **IService,** is a client library which calls the service host via Rest calls (you could also have ServiceRabbitMq etc). **ServiceRest** knows its own version number via reflection and appends the customer version number header with this information to each request

All three class libs are versioned, published on nuget and reference a specific version of Models via nuget. The reference should be set to ‘Specific version’ to enforce this. The version number of **Service** and **ServiceRest** should be synced, ie to call **Service** v 1.2.x.y you use **ServiceRest** v1.2.x.y

### Integration Tests

The Service solution contains integration tests for the Service. The Test project references the **SerivceRest** nuget package which it uses to call the service at an endpoint specified in app.config each method is called and the result compared to static test data. The ServiceInfo method is included, this will detect when the service is running a different version than that expected, or is incorrectly configured

When the service has a change the test project is duplicated and updated to use the new version of the client. You are then able to run both the old and new requests against the same endpoint to check for backwards compatibility. You are also able to change the endpoint to run the tests against INT/UAT/CUI/Production environments if required. Keep the Integration test when a new ServiceRest is created, or discard it if it’s identical (ie no breaking changes)

## Infrastructure

**Service** is hosted on two or more machines behind a load balancer exposing a single endpoint. The load balancer is capable of routing based on headers.

By delegating the routing of requests to the correct service version to a networking task, we enable a number of scenarios.

* Default to the latest/oldest/most popular version if no header is supplied
* Customer X wants to keep using the old version/upgrade to new version but have no budget to change their code. We know what IPs they are calling from and can route to the requested version.
* Legacy client X has to use an old version, but we don’t want to change the code to specify that version
* Legacy client X doesn’t happen to use the call which changed, so it can technically use the new version, but we don’t want to update the code, or support two versions of the Service
* We thought we updated everything, so why are we still getting some traffic to the old boxes?

## Deployment

When a new version of the service is deployed. New boxes are instanced, the service installed and started, the old boxes are taken out of load balance and the new ones added.

## Changes and version numbers

1. Changes to the data model, new fields etc

This requires the version of **Model** be incremented with in turn forces an update of both **Service** and **ServiceRest**

1. Changes to the service which break the interface, eg new methods

This requires a new version of the Interface, hence new  **Service** and **ServiceRest**

1. Changes to the client, new version of restsharp, bug fix etc

A non-breaking change can be limited to the **ServiceRest** version number updated but only a point release

1. Changes to the logic in the service which do not change the interface, ie bug fix

A non-breaking change can be limited to **Service** which gets a point release

As the Model is referenced by specific version, an application using **ServiceRest** will automatically be forced to use the correct version of the Model

As **ServiceRest** adds the version header, an application using it will automatically connect to the correct version of the service

As **ServiceRest** uses reflection to detect its version, it automatically picks up the assembly version number assigned by team city or however we set it.

As **ServiceRest** and **Sevice**  both implement **IService** and use **Model** most breaking changes in service, should force the creation of a new **ServiceRest,** you need to open the project, update your nuget packages, implement any changes to the interface and build.

As Octopus shows the deployed version number you can see what version is up in each environment and because the version numbers are synced, what version of **ServiceRest** your app should be using.

As each request includes the version number of **ServiceRest** you can tell what versions live applications are using via fiddler/network monitoring

**Non breaking changes**

Most changes will be non-breaking, bug fixes, tweaks and the like on the **Service**. These can be tested to ensure they are not breaking via the Integration tests as described above and deployed as normal.

**Breaking changes**

When a breaking change occurs, there will be a period where both old and new versions need to be up at the same time. Even if all clients are updated at the same time there will be a short period as servers are switched over.

In order to achieve this we deploy the new version to new servers as normal. The load balancer is then configured to route requests based on the version header to either the new or old servers. When no more traffic is being routed to the old servers then can be taken out of load.