**Four REST API Versioning Strategies**

At xMatters, we follow the [SemVer](http://semver.org/) specification – we update the API major version whenever we introduce breaking changes. Internally, we update minor and patch versions whenever we add functionality and backward-compatible updates. When we release a new major version of the [xMatters REST API](https://www.xmatters.com/features/workflow-process-automation), clients can choose to either continue using an existing major version or migrate to the new one.

One of the major challenges surrounding exposing services is handling updates to the API contract. Clients may not want to update their applications when the API changes, so a versioning strategy becomes crucial. A versioning strategy allows clients to continue using the existing REST API and migrate their applications to the newer API when they are ready.

There are four common ways to version a REST API.

**1. Versioning through URI Path**

http://www.example.com/api/**1**/products

One way to version a REST API is to include the version number in the URI path.

xMatters uses this strategy, and so do [DevOps teams](https://www.xmatters.com/use-cases/devops-sre) at Facebook, Twitter, Airbnb, and many more.

The internal version of the API uses the **1.2.3** format, so it looks as follows:

MAJOR.MINOR.PATCH

* **Major version:** The version used in the URI and denotes breaking changes to the API. Internally, a new major version implies creating a new API and the version number is used to route to the correct host.
* **Minor and Patch versions:** These are transparent to the client and used internally for backward-compatible updates. They are usually communicated in change logs to inform clients about a new functionality or a bug fix.

This solution often uses URI routing to point to a specific version of the API. Because cache keys (in this situation URIs) are changed by version, clients can easily cache resources. When a new version of the REST API is released, it is perceived as a new entry in the cache.

* **Pros:** Clients can cache resources easily
* **Cons:** This solution has a pretty big footprint in the code base as introducing breaking changes implies branching the entire API

**2. Versioning through query parameters**

http://www.example.com/api/products?version=1

Another option for versioning a REST API is to include the version number as a query parameter.

This is a straightforward way of versioning an API from an implementation point of view.

* **Pros:** It’s a straightforward way to version an API, and it’s easy to default to the latest version
* **Cons:** Query parameters are more difficult to use for routing requests to the proper API version

**3. Versioning through custom headers**

curl -H “Accepts-version: 1.0”  
http://www.example.com/api/products

REST APIs can also be versioned by providing custom headers with the version number included as an attribute.The main difference between this approach and the two previous ones is that it doesn’t clutter the URI with versioning information.

* **Pros:** It doesn’t clutter the URI with versioning information
* **Cons:** It requires custom headers

**4. Versioning through content negotiation**

curl -H “Accept: application/vnd.xm.device+json; **version=1**” http://www.example.com/api/products

The last strategy we are addressing is versioning through content negotiation.

This approach allows us to version a single resource representation instead of versioning the entire API which gives us a more granular control over versioning. It also creates a smaller footprint in the code base as we don’t have to fork the entire application when creating a new version. Another advantage of this approach is that it doesn’t require implementing URI routing rules introduced by versioning through the URI path.

One of the drawbacks of this approach is that it is less accessible than URI-versioned APIs: Requiring HTTP headers with media types makes it more difficult to test and explore the API using a browser.

* **Pros:** Allows us to version a single resource representation instead of versioning the entire API, which gives us a more granular control over versioning. Creates a smaller footprint. Doesn’t require implementing URI routing rules.
* **Cons:** Requiring HTTP headers with media types makes it more difficult to test and explore the API using a browser

**Summary**

Versioning is a crucial part of API design. It gives developers the ability to improve their API without breaking the client’s applications when new updates are rolled out.

# Versioning RESTful Web Services-Basic Approach With URIs

Versioning is the most important and difficult part of the API as it takes backward API compatible. Versioning helps us to iterate faster when the changes are identified. We should always version our Web API.

Consider a scenario in which we have a Web API that is up (status) and running. The users are consuming that API. Now we want to add more functionality in the Web API but want to keep the existing functionality unchanged. There may be few users who still want to use the old API while the other users want a new version of API with new or extended features. It is the scenario where Web API versioning comes into existence.

## When we required versioning:

When we made a **breaking change** in Web API, we should up versioned the API. Breaking changes includes:

* A change in the format of the response data for one or more calls.
* Change in the response type.
* Remove any part of the API.

Breaking changes should always result in a change to the major version number for an API or content response type.

Non-breaking changes (adding new points or new response parameters) do not require a change to the major version number. However, it can be helpful to track the minor version of the APIs.

## How to Version

The most commonly used approaches fall into three categories:

* URI Versioning
* Versioning using Custom Request Header
* Versioning using Accept Header

AD

### URI Versioning

URI versioning is the most straightforward approach. It specified in the URL as a query string. It violates the principle that a URI should refer to a unique resource. You are also guaranteed to break client integration when a version is updated. **Twitter** uses URI versioning.

**Example**

<http://api.demo.com/v1>  
[http://apiv1.demo.com](http://apiv1.demo.com/)

The version need not be numeric, nor specified using v[x] syntax. Alternatives include the date, project name, season, or other identifiers that are meaningful enough to change as the version change.

### Versioning using Custom Request Header

A custom header allows us to preserve our URLs. It is a duplicate of the content negotiation behavior implemented by the existing Accept header. Version information is specified in the Request Header without the need for any change in the URL. **Microsoft** uses the request header versioning. The user cannot access request header versioning in the normal browser (chrome). We are required a special plugin to access them on the browser.

**Example**

Accept-version: v1  
Accept-version: v2

### Versioning using Accept Header

Accept header define the media type and character encodings. We can also pass version information for Web API through accept headers without changing the URL. It is also known as media type versioning or content negotiation or accept header. Github uses the accept header versioning. The user cannot access accept-header versioning in the normal browser (chrome). We are required a special plugin to access them on the browser.

**Example**

Accept: application/vnd.demo.v1+json Accept:application/vnd.demo+json;version=1.0

Let's see how to implement versioning in the project.

### URI Versioning

**Step 1:** Create a class with the name **PersonV1.java** in the package **com.javatpoint.server.main.versioning**. PersonV1 denotes the first version of API. The initial version of API has a name variable.

**PersonV1.java**

1. **package** com.javatpoint.server.main.versioning;
2. **public** **class** PersonV1
3. {
4. **private** String name;
5. }

**Step 2:** Over a period, we recognize the need for having the first name and last name separately. So we created a class with the name **Person2.java**. It denotes the second version of API.

**PersonV2.java**

1. **package** com.javatpoint.server.main.versioning;
2. **public** **class** PersonV2
3. {
4. **private** Name name;
5. }

**Step 3:** Create a class with the name **Name.java** that has two variables **firstName** and lastName separately.

AD

**Name.java**

1. **package** com.javatpoint.server.main.versioning;
2. **public** **class** Name
3. {
4. **private** String firstName;
5. **private** String lastName;
6. }

The old version is still returning the full name, and the second version is returning firstName and lastName separately. Now we are required to create two different versions of the same service.

Let's see how to create two different versions of the same service and what are the different versions of the service are present.

**Step 4:** In the **Name.java** file, Generate Getters and Setters, Generate Constructor using Fields. Create a no-argument constructor of the class Name.

**Name.java**

1. **package** com.javatpoint.server.main.versioning;
2. **public** **class** Name
3. {
4. **private** String firstName;
5. **private** String lastName;
6. //no argument constructor
7. **public** Name()
8. {
10. }
11. **public** Name(String firstName, String lastName)
12. {
13. **super**();
14. **this**.firstName = firstName;
15. **this**.lastName = lastName;
16. }
17. **public** String getFirstName()
18. {
19. **return** firstName;
20. }
21. **public** **void** setFirstName(String firstName)
22. {
23. **this**.firstName = firstName;
24. }
25. **public** String getLastName()
26. {
27. **return** lastName;
28. }
29. **public** **void** setLastName(String lastName)
30. {
31. **this**.lastName = lastName;
32. }
33. }

**Step 5:** Open **PersonV1.java** class. Generate Getters and Setters, Generate Constructor using Fields. Create a no argument constructor of the class PersonV1.java.

**PersonV1.java**

1. **package** com.javatpoint.server.main.versioning;
2. **public** **class** PersonV1
3. {
4. **private** String name;
5. //no argument constructor
6. **public** PersonV1()
7. {
8. **super**();
9. }
10. **public** PersonV1(String name)
11. {
12. **super**();
13. **this**.name = name;
14. }
15. **public** String getName()
16. {
17. **return** name;
18. }
19. **public** **void** setName(String name)
20. {
21. **this**.name = name;
22. }
23. }

**Step 6:** Open PersonV2.java. Generate Getters and Setters, Generate Constructor using Fields. Create a no argument constructor of the class PersonV2.java.

**PersonV2.java**

1. **package** com.javatpoint.server.main.versioning;
2. **public** **class** PersonV2
3. {
4. **private** Name name;
5. **public** PersonV2()
6. {
7. **super**();
8. }
9. **public** PersonV2(Name name)
10. {
11. **super**();
12. **this**.name = name;
13. }
14. **public** Name getName()
15. {
16. **return** name;
17. }
18. **public** **void** setName(Name name)
19. {
20. **this**.name = name;
21. }
22. }

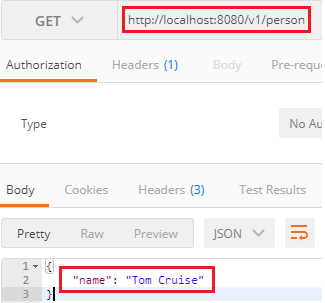
Now we need to create a service.

**Step 7:** Create a class with the name **PersonVersioningController.java**. Create two methods for different versions and map them to different URIs.

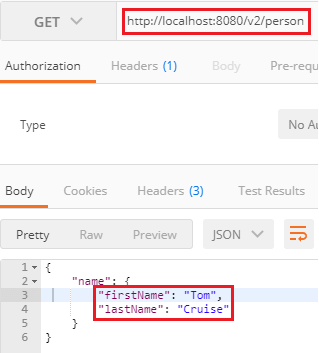
**PersonVersioningController.java**

1. **package** com.javatpoint.server.main.versioning;
2. **import** org.springframework.web.bind.annotation.RestController;
3. **import** org.springframework.web.bind.annotation.GetMapping;
4. @RestController
5. **public** **class** PersonVersoningController
6. {
7. //this method is for the first version that returns the entire name
8. @GetMapping("v1/person")
9. **public** PersonV1 personv1()
10. {
11. **return** **new** PersonV1("Tom Cruise");
12. }
13. //this method is for the second version that returns firstName and lastName separately
14. @GetMapping("v2/person")
15. **public** PersonV2 personv2()
16. {
17. **return** **new** PersonV2(**new** Name("Tom", "Cruise"));
18. }
19. }

**Step 8:** Open the **Postman** and send a **GET** request with the URI http://localhost:8080/v1/person. It returns the full name, as shown in the following image.



Change the URI http://localhost:8080/v2/person for the second version. It returns the firstName and lastName separately, as shown in the following image.



## Versioning using Request Parameter

Another way to implement versioning is by using the request parameter. Amazon uses the request parameter versioning. Open the PersonVersioningController.java and do the following changes:

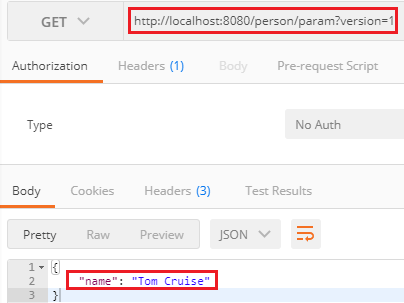
* Change the URI for the first method from /v1/person to /person/param.
* Change the name of the method from personV1 to paramV1.
* Similarly, change the URI for the second method from /v2/person to /person/param.

Both the methods have the same get mapping, so we will distinguish them by using the value and params attribute. The value attribute contains the URI, which we want to use, and the params attribute contains the parameter which distinguishes between versions.

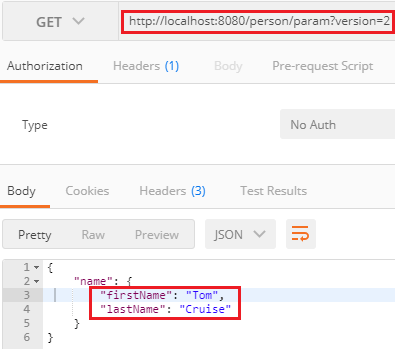
**PersonVersoningController.java**

1. **package** com.javatpoint.server.main.versioning;
2. **import** org.springframework.web.bind.annotation.GetMapping;
3. **import** org.springframework.web.bind.annotation.RestController;
4. @RestController
5. **public** **class** PersonVersoningController
6. {
7. //this method is for first version that returns the entire name
8. @GetMapping(value="/person/param", params="version=1")
9. **public** PersonV1 personV1()
10. {
11. **return** **new** PersonV1("Tom Cruise");
12. }
13. //this method is for second version that returns firstName and lastName separately
14. @GetMapping(value="/person/param", params="version=2")
15. **public** PersonV2 personV2()
16. {
17. **return**  **new** PersonV2(**new** Name("Tom", "Cruise"));
18. }
19. }

Now, move to Postman and send a GET request with the URI http://localhost:8080/person/param?version=1. It returns the full name, as shown in the following image.



Again, generate a GET request with the URI http://localhost:8080/person/param?version=2 to access the second version. It returns the firstName and lastName separately, as shown in the following image.



### Versioning using Request Header

There is another option to do versioning using the Request Header. It is similar to content negotiation. In this method, we differentiate service based on the Request Header.

In the **PersonVersioningController.java**, do the following:

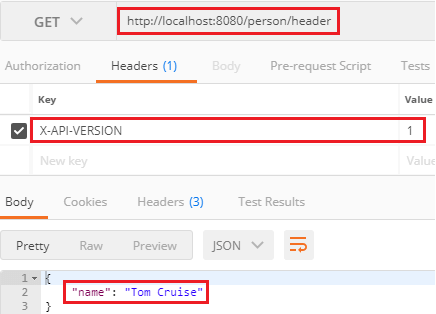
* Copy both the methods and paste in the same file.
* Change the method name paramV1 to headerV1 and paramV2 to headerV2.
* Replace the URI /person/param with /person/header and params with headers.

1. /\*---------------using request header--------------\*/
2. //this method is for first version that returns the entire name
3. @GetMapping(value="/person/header", headers="X-API-Version=1")
4. **public** PersonV1 headerV1()
5. {
6. **return** **new** PersonV1("Tom Cruise");
7. }
8. //this method is for second version that returns firstName and lastName separately
9. @GetMapping(value="/person/header", headers="X-API-Version=2")
10. **public** PersonV2 headerV2()
11. {
12. **return** **new** PersonV2(**new** Name("Tom", "Cruise"));
13. }

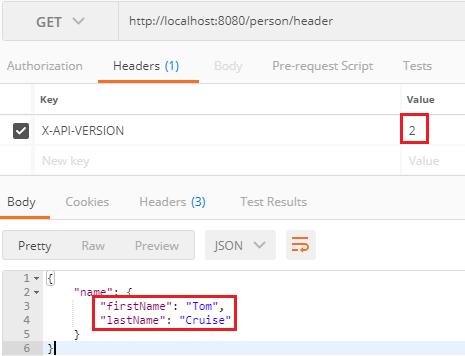
Open the **Postman**:

* Select the Headers tab and set key: X-API-VERSION and Value: 1.
* Type the URI http://localhost:8080/person/header and send a GET request.

It returns the name full name.



Let's send a GET request for version 2. For this, we need to change the value from 1 to 2 under the Headers tab. It returns the firstName and lastName separately.



AD

## Versioning using Accept Header

Another method that is used in versioning is the Accept Header. It is also known as Content Negotiation or Accept Versioning. In this method, we use an attribute called produce. It denotes what kind of output we are generating for the specific service.

In the **PersonVersioningController.java** file, do the following:

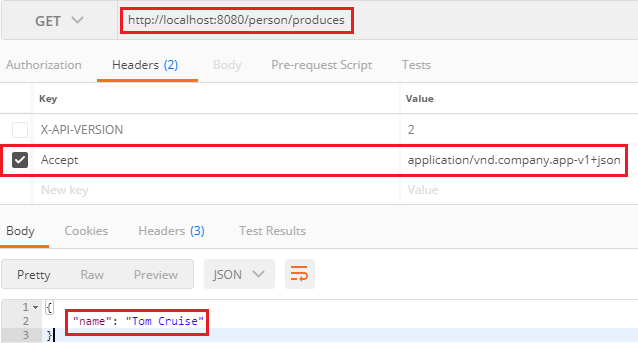
* Copy both the methods and paste in the same file.
* Change the methods name headerV1 to producesV1 and headerV2 to ProducesV2.
* Replace the URI /person/header with /person/produces and header with produces.

1. /\*---------------using accept header--------------\*/
2. //this method is for first version that returns the entire name
3. @GetMapping(value="/person/produces", produces="application/vnd.company.app-v1+json")
4. **public** PersonV1 producesV1()
5. {
6. **return** **new** PersonV1("Tom Cruise");
7. }
8. //this method is for second version that returns firstName and lastName separately
9. @GetMapping(value="/person/produces", produces="application/vnd.company.app-v2+json")
10. **public** PersonV2 producesV2()
11. {
12. **return**  **new** PersonV2(**new** Name("Tom", "Cruise"));
13. }

Open the **Postman**:

* Select the Headers tab and set key: Accept and Value: application/vnd.company.app-v1+json.
* Uncheck the X-API-VERSION key.
* Type the URI http://localhost:8080/person/produces and send a GET request.

It returns the full name.



Let's send a GET request for version 2. For this we need to change the value from **Value: application/vnd.company.app-v1+json** to **Value: application/vnd.company.app-v2+json**.

It returns the firstName and lastName separately.

