Instructor Notes:

Add instructor notes here.

Instructor Notes:

Lesson Objectives

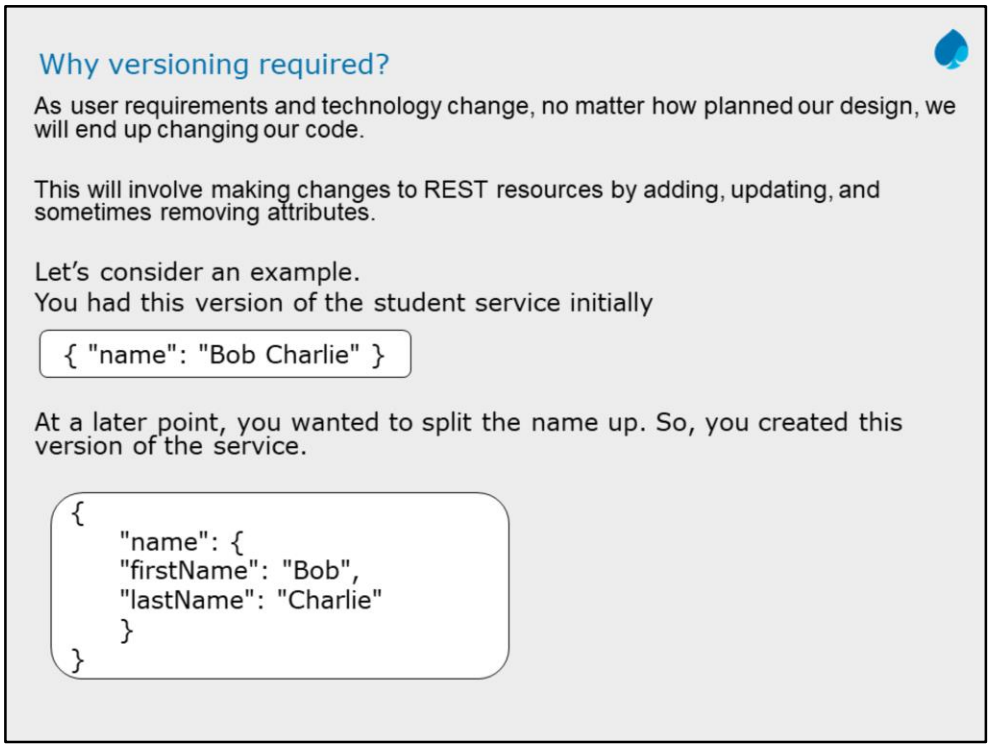
Introduction to Spring MVC framework

- Why versioning required?
- Versioning Approaches
  - URI versioning
  - URI parameter versioning
  - Accept header versioning
  - Custom header versioning

References:

http://www.springboottutorial.com/spring-boot-versioning-for-rest-services
https://www.safaribooksonline.com/library/view/spring-rest/9781484208236/9781484208243_Ch07.xhtml

Instructor Notes:



As user requirements and technology change, no matter how planned our design, we will end up changing our code. This will involve making changes to REST resources by adding, updating, and sometimes removing attributes. Although the crux of the API—read, create, update, and remove one or more resources—remains the same, this could result in such drastic changes to the representation that it may break any existing consumers. Similarly, changes to functionality such as securing our services and requiring authentication or authorization can break existing consumers. Such major changes typically call for new versions of the API

Let's consider an example.

You had this version of the student service initially

{ "name": "Bob Charlie" }

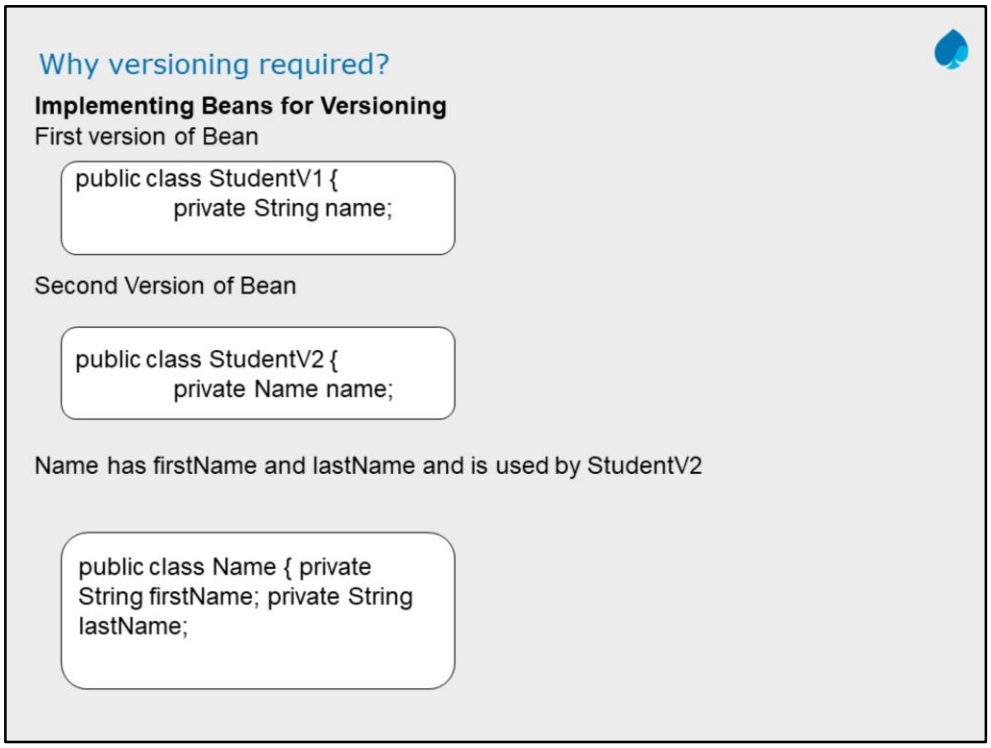
At a later point, you wanted to split the name up. So, you created this version of the service.

{ "name": { "firstName": "Bob", "lastName": "Charlie" } }

You can support both these requests from the same service, but it becomes complex as the requirements diversify for each of the versions.

In these kind of situations, versioning becomes mandatory.

Instructor Notes:



Implementing Beans for Versioning

```
public class StudentV1 {
                private String name;
                . . .
}
```

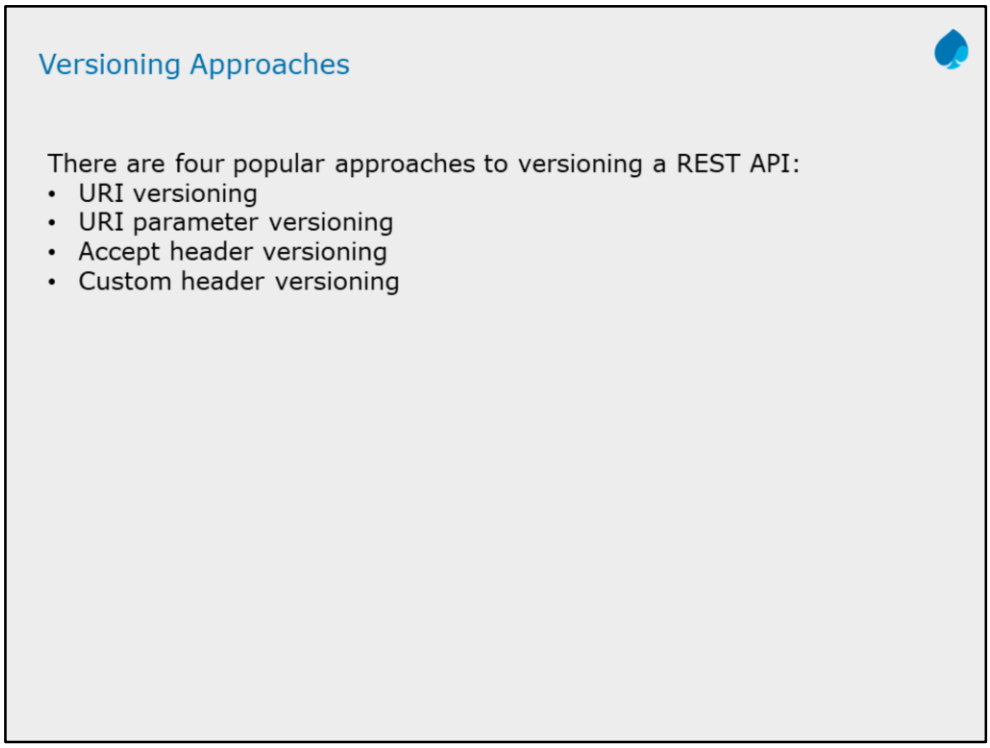
Second Version of Bean

```
public class StudentV2 {
                private Name name;
                . . .
}
```

Name has firstName and lastName and is used by StudentV2

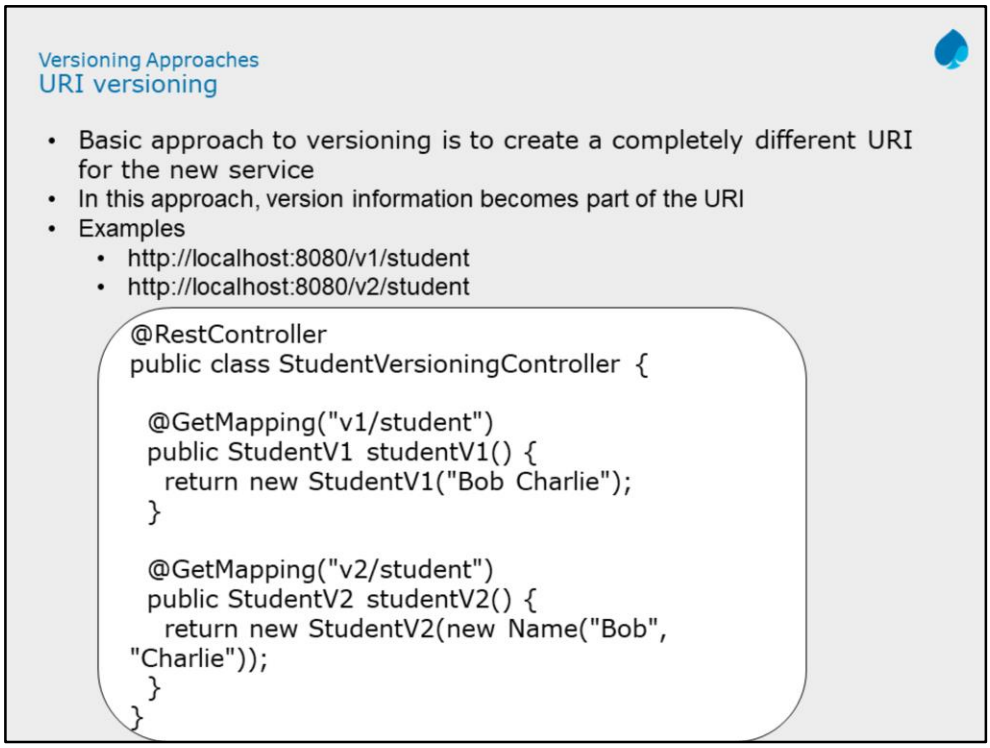
```
public class Name {
                private String firstName;
                private String lastName;
                . . .
}
```

Instructor Notes:



There are four popular approaches to versioning a REST API:
URI versioning
URI parameter versioning
Accept header versioning
Custom header versioning
None of these approaches are silver bullets and each has its fair share of advantages and disadvantages.

Instructor Notes:



In this approach, version information becomes part of the URI. For example, http://api.example.org/v1/users and http://api.example.org/v2/users r epresent two different versions of an application API. Here we use v notation to denote versioning and the numbers 1 and 2 following the v indicate the first and second API versions.
URI versioning has been one of the most commonly used approaches and is used by major public APIs such as Twitter, LinkedIn, Yahoo, and SalesForce. Here are some examples:
LinkedIn: https://api.linkedin.com/v1/people/~
Yahoo: https://social.yahooapis.com/v1/user/12345/profile
SalesForce: http://na1.salesforce.com/services/data/v26.0
Twitter: https://api.twitter.com/1.1/statuses/user_timeline.json
Twilio: https://api.twilio.com/2010-04-01/Accounts/{AccountSid}/Calls
As you can see, LinkedIn, Yahoo, and SalesForce use the v notation. In addition to a major version, SalesForce uses a minor version as part of its URI version. Twilio, by contrast, takes a unique approach and uses a timestamp in the URI to differentiate its versions.

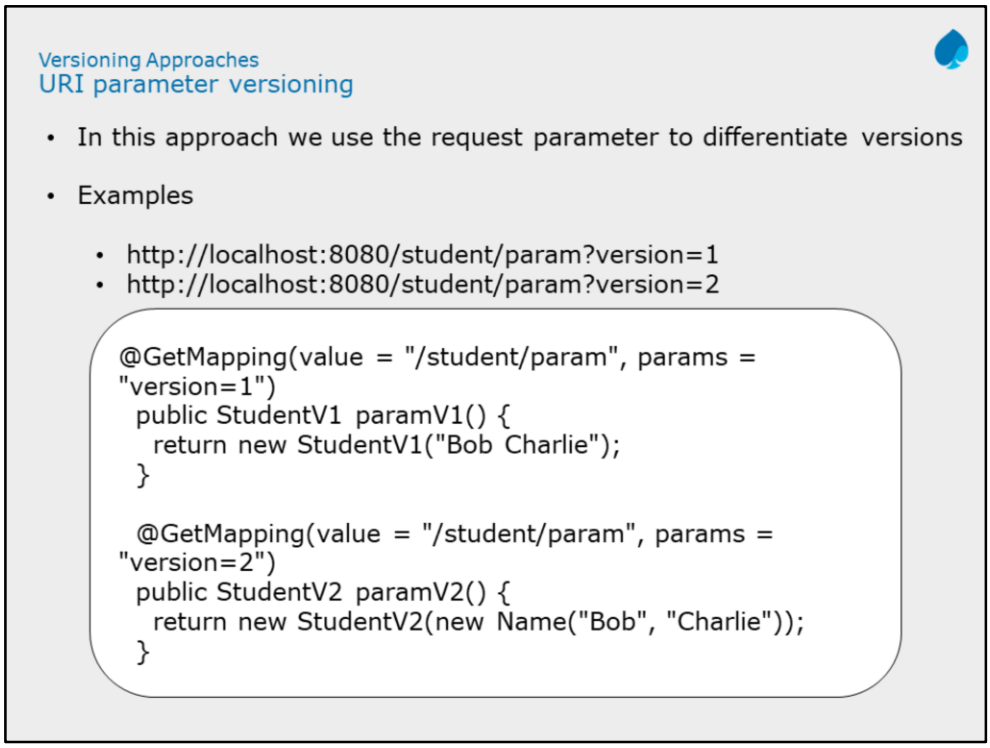
Instructor Notes:

These demos can be executed for better understanding

URI Versioning Demo

SpringRESTURIVersioningDemo

Instructor Notes:



This is similar to the URI versioning that we just looked at except that the version information is specified as a URI request parameter. For example, the URI http://api.example.org/users?v=2 uses the version parameter v to represent the second version of the API. The version parameter is typically optional and a default version of the API will continue working for requests without version parameter. Most often, the default version is the latest version of the API.
Although as not popular as other versioning strategies, a few major public APIs such as Netflix have used this strategy. The URI parameter versioning shares the same disadvantages of URI versioning. Another disadvantage is that some proxies don't cache resources with a URI parameter, resulting in additional network traffic.

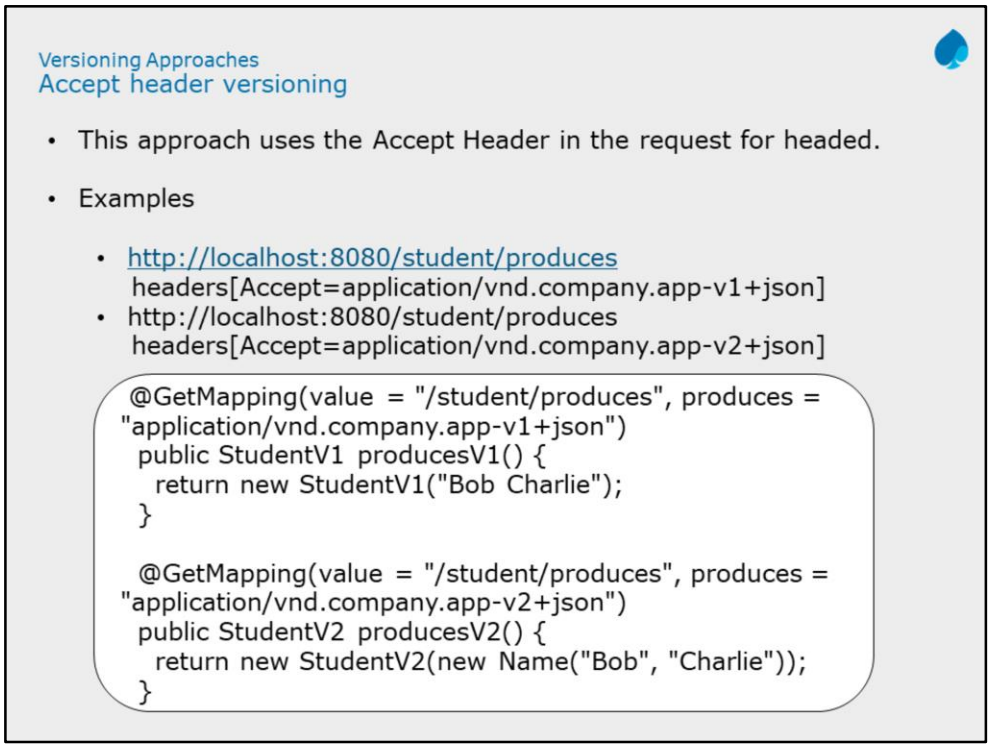
Instructor Notes:

These demos can be executed for better understanding

URI Parameter Versioning Demo

SpringRESTURIParameterVersioningDemo

Instructor Notes:



This versioning approach uses the Accept header to communicate version information. Because the header contains version information, there will be only one URI for multiple versions of API.
Up to this point, we have used standard media types such as "application/json" as part of the Accept header to indicate the type of content the client expects. To pass additional version information, we need a custom media type. The following convention is popular when creating a custom media type:
vnd.product_name.version+ suffixThe vnd is the starting point of the custom media type and indicates vendor. The product or producer name is the name of the product and distinguishes this media type from other custom product media types. The version part is represented using strings such as v1 or v2 or v3. Finally, the suffix is used to specify the structure of the media type. For example, the +json suffix indicates a structure that follows the guidelines established for media type "application/json". RFC 6389 (https://tools.ietf.org/html/rfc6839) gives a full list of standardized prefixes such as +xml, +json, and +zip. Using this approach, a client, for example, can send an application/vnd.quickpoll.v2+json accept header to request the second version of the API.
The Accept header versioning approach is becoming more and more popular as it allows fine-grained versioning of individual resources without impacting the entire API. This approach can make browser testing harder as we have to carefully craft the Accept header. GitHub is a popular public API that uses this Accept header strategy. For requests that don't contain any Accept header, GitHub uses the latest version of the API to fulfill the request.

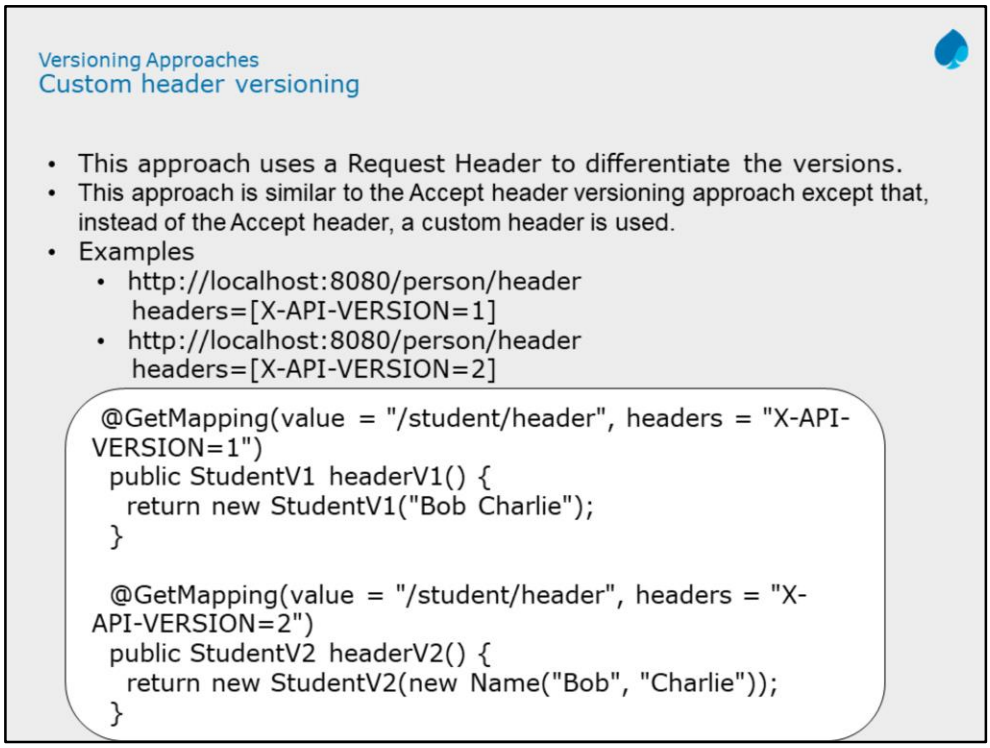
Instructor Notes:

These demos can be executed for better understanding

Accept Header Versioning Demo

SpringRESTAcceptHeaderVersioningDemo

Instructor Notes:



The custom header versioning approach is similar to the Accept header versioning approach except that, instead of the Accept header, a custom header is used. Microsoft Azure takes this approach and uses the custom header x-ms-version. For example, to get the latest version of Azure at the time of writing this book, your request needs to include a custom header:
x-ms-version: 2014-02-14
This approach shares the same pros and cons as that of the Accept header approach. Because the HTTP specification provides a standard way of accomplishing this via the Accept header, the custom header approach hasn't been widely adopted.

Instructor Notes:

These demos can be executed for better understanding

Custom Header Versioning Demo

SpringRESTCustomHeaderVersioningDemo

Instructor Notes:

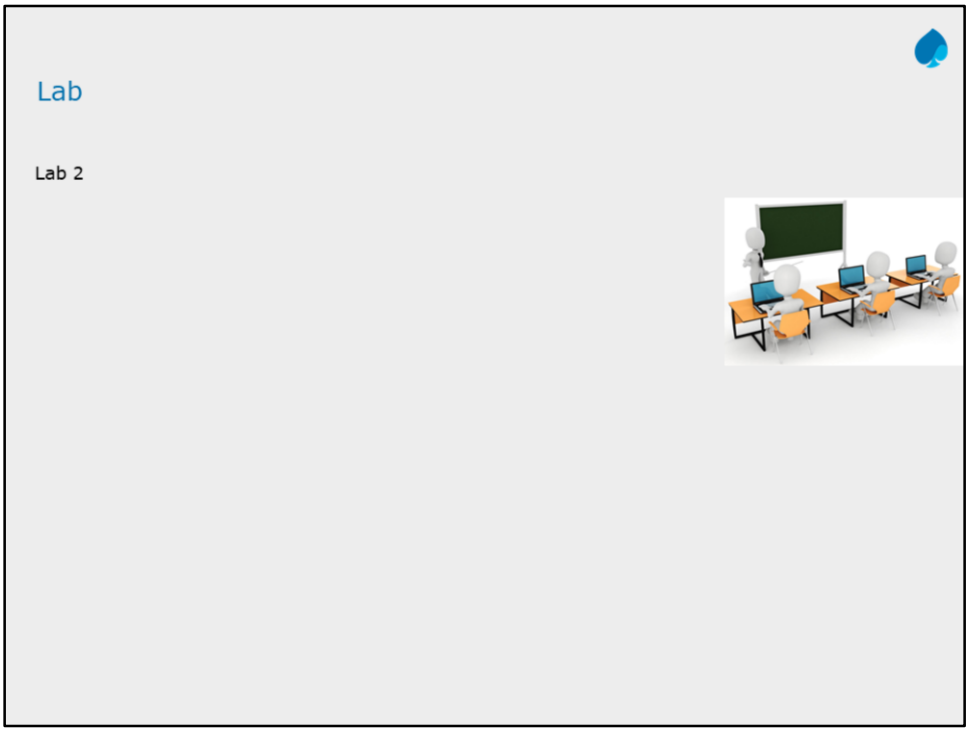
Summery

What we have seen so far:

References:
https://www.genuitec.com/spring-frameworkrestcontroller-vs-controller/

Instructor Notes:

Corresponding lab assignment

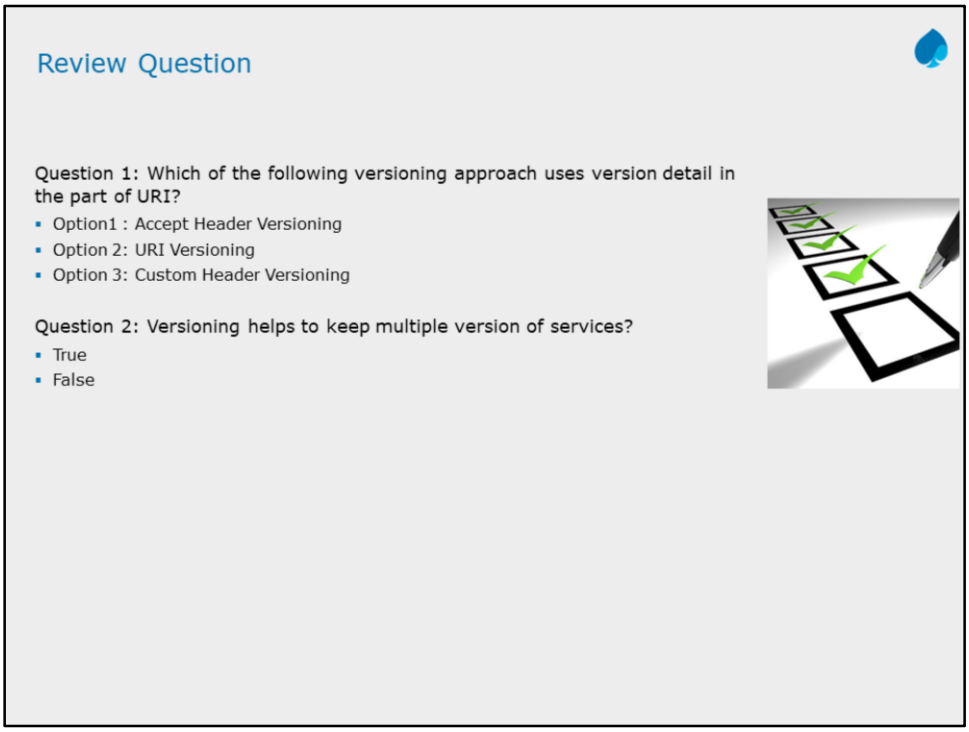


Lab number will be added after lab book will be ready

Instructor Notes:

Question 1: Option 2

Question 2: True



Add the notes here.