DEVCOR 350-901 Study Guide

Constantin Mohorea
CCIE® No. 16223, CCDE® No. 20170054, DevNet 500

Cisco Press

Cisco DEVCOR 350-901 Study Guide

Edition: 2021.3

Constantin Mohorea

Cisco Press

Cisco DEVCOR 350-901 Study Guide

Constantin Mohorea

Published by: Cisco Press

ISBN-13: 978-0-13-750004-8

ISBN-10: 0-13-750004-1

Warning and Disclaimer

This book is designed to provide information about the “Cisco Certified DevNet Specialist – Core” DEVCOR 350-901 exam. Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied.

The information is provided on an “as is” basis. The author, Cisco Press, and Cisco Systems, Inc., shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of programs that may accompany it.

The opinions expressed in this book belong to the authors and are not necessarily those of Cisco Systems, Inc.

Trademark Acknowledgments

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Cisco Press or Cisco Systems, Inc., cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at

corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

About the Author

Constantin Mohorea is a CX Consulting Engineer at Cisco and is an experienced network professional with a demonstrated history of helping clients achieve their business goals. He excels at building relationships with client teams, understanding their unique requirements, and delivering innovative, stable, and effective network solutions. He brings more than 20 years of experience in the IT industry, specializing in network design, enterprise, security, and data center technologies for customers across a wide spectrum of industries. He is a Cisco Certified Design Expert (CCDE# 20170054) and double Cisco Certified Internetwork Expert (CCIE# 16223) – Security and Enterprise Infrastructure (formerly Routing and Switching).

Constantin has long been interested in programming in general and was happy to see how the networking industry was evolving toward programmability and automation. He was excited about the Cisco DevNet program and got certified on the very first day, earning the honorary DevNet 500 designation. He is a Cisco Certified DevNet Professional and is looking forward to becoming a Cisco Certified DevNet Expert when this certification becomes available.

About the Technical Reviewer

Dmitry Figol, CCIE No. 53592 (R&S) and Cisco Certified DevNet Professional, is a network automation architect at Cisco Global Demo Engineering. He is in charge of software architecture design and implementation of network automation systems. His main expertise is network programmability and Python. Previously, Dmitry worked in Cisco Sales as well as on the Technical Assistance Center (TAC) Core Architecture and VPN teams. Dmitry is a regular conference speaker. He also does live streams on Twitch about network automation and hosts the Network Automation Hangout audio show. Dmitry holds a Bachelor of Science degree in telecommunications. Dmitry can be found on Twitter as @dmfigol.

Dedication

I dedicate this book to my family: my parents, my wife, and especially my children. Thank you for all your support!

Special thanks go to the creators and members of the RouterGods networking community.

In addition, I'd like to dedicate this book to all the people who keep studying and improving themselves: Don't stop!

Table of Contents

Table of Figures

2.3 Identify ways to optimize API usage through HTTP cache controls

When API load increases due to growth of user requests and application data, hardware requirements increase as well. There are a few ways to address this: you can scale hardware (horizontally or vertically) or optimize your application, or both. The goals of the application optimizations are as follows:

- Improve response time in user-interactive applications.
- Decrease the required bandwidth.
- Decrease processing times.
- Lower resource utilization.

These goals may be achieved with pagination (see Section 2.4), caching, and compression.

HTTP Caching

HTTP caching is a technique that stores copies of web resources and serves them back when requested. Typically limited to caching responses to "GET" requests, HTTP caches intercept these requests and return local copies instead of redownloading resources from originating servers.

Caching may be implemented at different places: at the client (for example, in a web browser), locally as an organization's proxy server, on the server side as a standalone reverse proxy, or as a part of an API gateway.

Caching reduces the amount of network traffic and the load of web servers. It also improves the user experience since HTTP requests may be completed in less time. However, resources may change, and it is important to cache them properly: only until they change, not longer. To address this, caches use the concept of "freshness."

A stored resource is considered "fresh" if it may be served directly from the cache. As web servers cannot contact caches and clients when a resource changes, they simply indicate an expiration time for the provided content. Before this expiration time, the resource is "fresh," and after the expiration time, the resource is "stale."

"Stale" content is not ignored and might still be reused, but it needs to be validated first. During validation, a check is performed on the original resource to verify that the cached copy is still current. Such checks usually involve only HTTP headers (HTTP "HEAD" request) and no data, so they are "cheap" compared to retrieving the whole resource. Validation checks increase the response time; however, they are more reliable than a time-based mechanism because they ensure data is up to date.

Website content may change when the cache is still "fresh," so some servers may want to force validation on every request to avoid inconsistencies. One way to achieve that is to set the explicit expiration time in the past and thus indicate that the response is already stale when it's received.

Validation can only happen if the origin server previously provided a *validator* for a resource, which is some value that describes a specific version of a resource. There are two types of validators:

- The date of last modification of the document, provided in the "Last-Modified" HTTP header.
- Some string that uniquely identifies a version of a resource, called the Entity Tag (ETag) and provided in the "ETag" HTTP header. It is not specified how ETag values are generated, so it may be a hash of the content, a hash of the last modification timestamp, or just a revision number.

Validators may be *strong* when they guarantee that two resources are completely byte-to-byte identical to each other. With the *weak* validators, two versions of the document are considered as identical if the content is equivalent (for example, same web page but with the different ads on it). ETags may be used as strong validators (by default) or as weak ones (prefixed with "W/"; for example, W/"5bfb1b822b0f7").

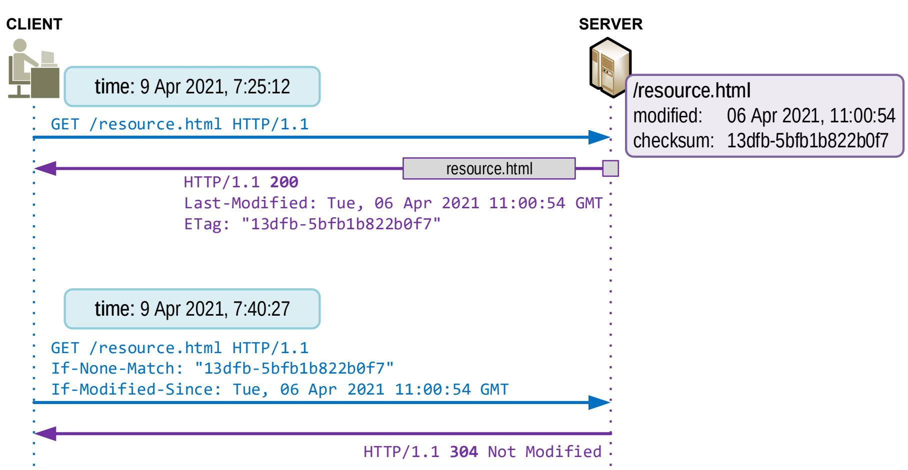
HTTP Conditional Requests

With HTTP *conditional requests,* servers' responses are based on the result of a comparison between the current version of a resource with validators included in a request. Conditional requests enable an efficient validation process but have other uses as well: to verify the integrity of a document (for example, when resuming a download) or, when clients act in parallel, to prevent accidental deletion or overwriting of one version of a document with another.

Special headers are used to specify preconditions in requests, and their logic depends on the HTTP request type: safe (read) or unsafe (write), as presented in Figure 26. In these examples, the "Last-Modified" and "ETag" headers are already known from previous requests.

With safe methods (GET and HEAD), conditions are used to fetch a resource *only when needed.*

Figure 26: HTTP Conditional Requests: Fetch the Resource



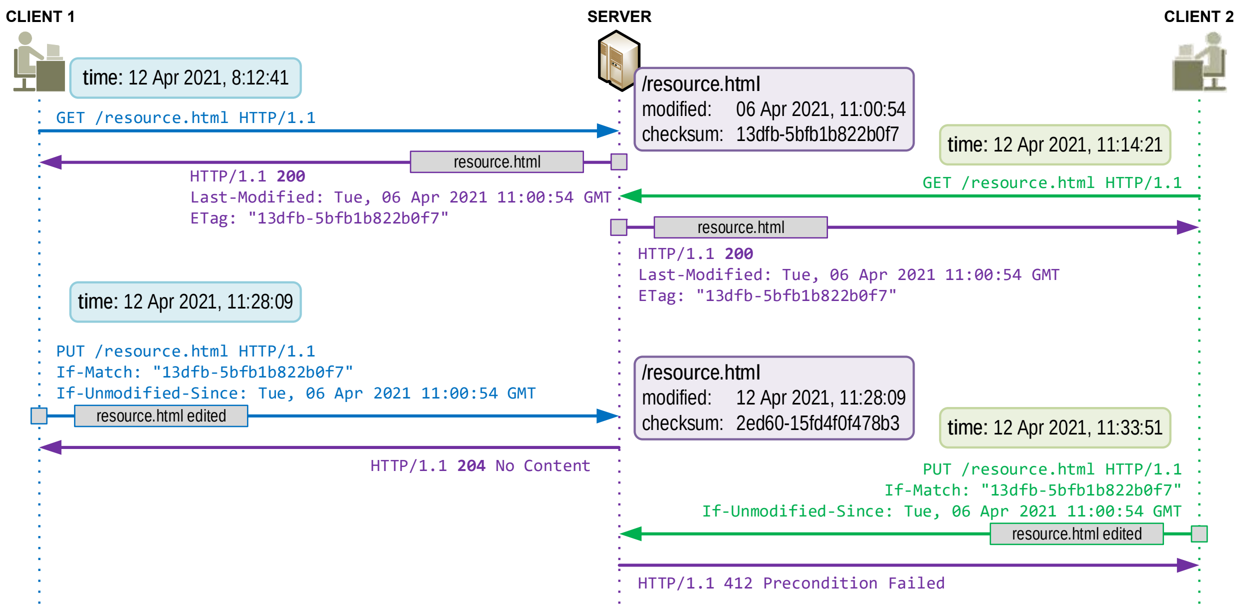
The following headers specify a condition for the request:

- **"If-None-Match"**: Set to previously known "ETag" value. The condition is true if the ETag of the resource on a server is different from the one given in this header. Both strong and weak ETags may be used.
- **"If-Modified-Since"**: Set to previously known "Last-Modified" value. The condition is true if the date of the resource on a server is more recent than the one listed in this header.

If the condition is true (that is, the resource has changed), then the server responds with the HTTP 200 code and the complete resource (for the GET method). If the resource hasn't changed, the server must respond with the HTTP status code 304 "Not Modified" and an empty body.

With unsafe methods (PUT and DELETE), conditions are used to allow modifications to a resource *only when it's still the same and hasn't changed since the last interaction* (see Figure 27).

Figure 27: HTTP Conditional Requests: Modify the Resource



The following headers specify a condition for the request:

- **"If-Match"**: Set to previously known "ETag" value. The condition is true if the ETag of the resource on a server is equal to the one supplied in this header. Only a strong ETag may be used, as weak tags never match under this comparison.
- **"If-Unmodified-Since"**: Set to previously known "Last-Modified" value. The condition is true if the date of the resource on a server is the same or older than the one listed in this header.

If the condition is true (that is, the resource has not changed), then the server proceeds with the update as normal. If the condition is not met, then the change is rejected with a 412 "Precondition Failed" error.

Caching Controls

Sometimes, caching may interfere with the application behavior, and the server administrator may wish to provide explicit directions to cache engines. The *Cache-Control* HTTP header is used for this purpose. Some of the most often used options are:

- **no-store***:* The response may not be stored in any cache; disable caching.
- **no-cache**: Allow caching, but stored response *must* be validated first before using it.
- **max-age***:* Time in seconds a cached copy is valid. `max-age=0` is similar to `no-cache`.
- **public***:* Can be cached by anyone.

- **private**: Only browser/client may cache.

Example: `Cache-Control: no-cache, max-age=0`

HTTP Data Compression

By using HTTP compression, less data needs to be transferred over the network, which results in faster response times and lower bandwidth utilization.

When making an HTTP request, a client may include an HTTP "Accept-Encoding" header to indicate a list of compression algorithms that the client supports. Common examples are compress, deflate, gzip, and bzip2.

Compression is not always accepted by the server, as it may cause issues with antivirus software, next-gen firewalls, or proxies, and/or it may violate company policies.

When the server receives a request that indicates compression support, it can honor that request or ignore it and send an uncompressed response. If one of the client's compression methods is supported, and the server decides to use it to compress the response, the server must include it in the "Content-Encoding" HTTP header.

2.4 Construct an application that consumes a REST API that supports pagination

When a client makes a REST API request, the response can be very large (for example, event log). Quite often, the client is not interested in all the data. For example, it might only need log entries for yesterday or the 50 latest chat messages. Pagination is a method to split resulting data into manageable chunks to allow better handling of large data sets and faster request processing.

The main reasons for using pagination in REST APIs are as follows:

- **To improve response times and end-user experience**: Because paginated responses are much smaller, they can be handled by the server and the network faster, and faster responses provide better user experiences.
- **To save resources**: Providing small responses demands much less compute and network resources.

One of the simplest types of pagination is the "Offset/Page"-based pagination. With this method, the client supplies the data offset and data size as query parameters. Here are two examples:

- **https://example.com/events?offset=100&limit=20**: Returns 20 entries, starting with 100th
- **https://example.com/events?page=6&per_page=20**: Same logic, different syntax

Typically, if offset or limit parameters are omitted, the server will use defaults (for example, offset=0 and limit=50).

Offset-based pagination is simple to implement and to use; however, it has its downsides:

- Adding or deleting entries (known as *page drift*) between calls may cause confusion. Some results may be skipped over (for example, some item is deleted and now the first item on a new page moved to the previous) and some may be included more than once (when items are inserted).