B2C HEADLESS COMMERCE IMPLEMENTATION GUIDE

Contents

1 Introduction

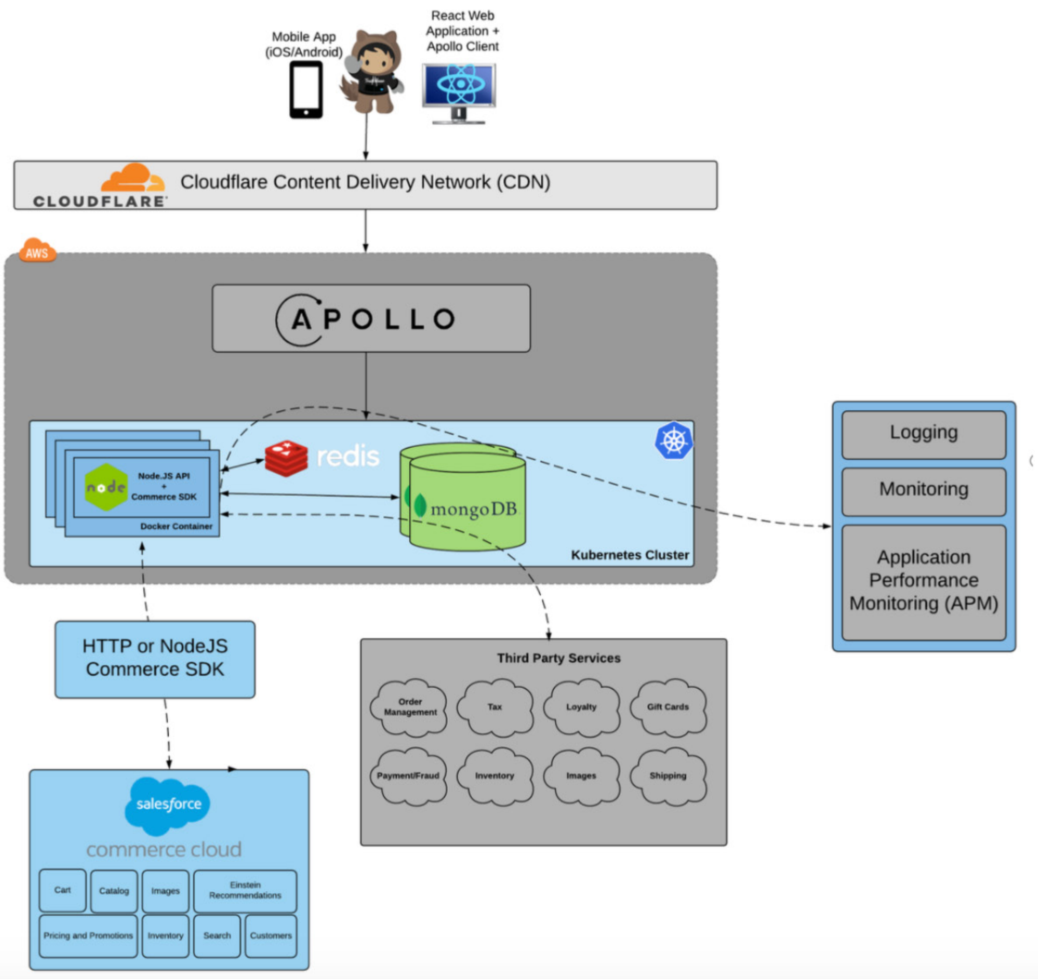
If you are considering a headless commerce solution with Salesforce B2C Commerce or have already chosen this approach, this guide is for you.

This paper will cover how to approach essential aspects of developing a solution to headless ecommerce using Salesforce B2C Commerce APIs. The guidance focuses on designing for performance and scale, with a sample architecture provided for context.

We strongly recommend reading Getting Started with Headless Commerce first to get familiar with many of the topics covered in more detail in this paper. We also recommend visiting the Salesforce Commerce Cloud Developer Center **(CCDC)** for documentation on the Salesforce B2C Commerce APIs, solution kits, and sample apps to better understand our headless product offerings.

2 Sample Architecture

At Salesforce B2C Commerce, we believe in a back-end for front-end (BFF) architecture approach to headless ecommerce. Having a BFF layer custom-tailored for each user experience allows you to design your back end to support the use cases of your front end in a way that can perform and scale to suit growing demands for your brand. Designing a BFF layer reduces direct chatter between the client and back-end systems, and allows your front-end and back-end teams to work independently while delivering new functionality. The architecture depicted below is meant to serve as a point of reference for the topics described in this paper. This should not be confused with a recommended or required architecture, as your architecture will vary depending on the requirements for your headless implementation.



Below is a description of each component involved in the sample architecture:

ReactJS

ReactJS is a component-based front-end JavaScript library for designing features of your UI, often those that require interaction with back-end systems.

Content Delivery Network (CDN)

The role of the CDN is to cache and serve static content to the user in order to reduce the pressure on downstream systems. This significantly reduces page load times, which improves the user experience and higher order conversation rates. Caching static content will allow your application to only do processing that requires retrieving real-time data. The CDN deploys Point of Presence (PoP) servers around the world to ensure your geographically distributed user base is always receiving cached content from the nearest PoP. Most CDNs also provide other performance optimization features we will discuss in more detail in a later section.

Apollo Client

The Apollo client within your front end is the single point of entry into the underlying back-end systems that comprise your ecommerce solution. The Apollo client retrieves data from the Apollo Server via GraphQL queries over HTTP.

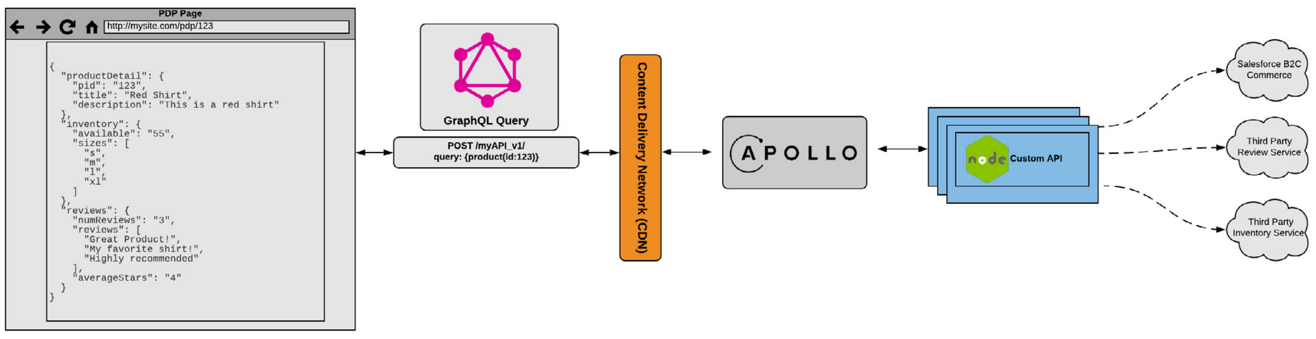
Apollo Server

The Apollo Server is a graph-based platform for connecting clients to your back-end services using a simple query language called GraphQL. The Apollo server will be the front end's single point of entry into multiple underlying systems that comprise your ecommerce solution, for example, a third-party REST API that calculates taxes for a customer cart, or a database that contains information about a customer. The Apollo Server provides several benefits to your architecture. It:

- Reduces the number of requests needed between the front end and underlying back-end systems
- Abstracts back-end systems from front-end developers, requiring them to learn just one API
- Offers fine-grained cache control for improved performance
- Has built-in monitoring and logging for execution of GraphQL operations

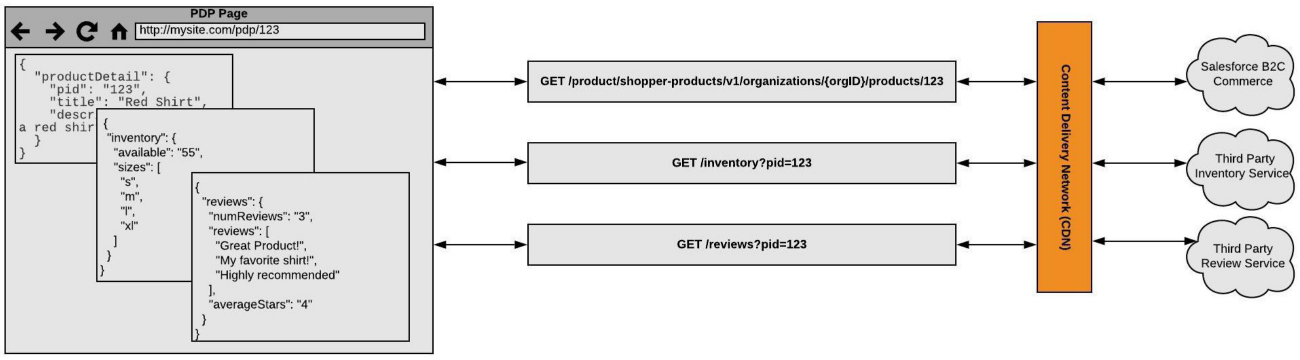
Below is an example of how using an Apollo Server provides a benefit over a direct client-to-service approach. In the second diagram depicting the latter, the client has to assemble a response based on three separate requests, while the Apollo Server approach makes a single call to one endpoint that does the heavy lifting and returns a single response back to the client.

Apollo PDP Request


PDP Page
http://mysite.com/pdp/123
GraphQL Query
POST /myAPI_v1/
query: {product(id:123)}
Content Delivery Network (CDN)
APOLLO
node
Custom API
Salesforce B2C Commerce
Third Party Review Service
Third Party Inventory Service

```
{
  "productDetail": {
    "pid": "123",
    "title": "Red Shirt",
    "description": "This is a red shirt"
  },
  "inventory": {
    "available": "55",
    "sizes": [
      "s",
      "m",
      "l",
      "xl"
    ]
  },
  "reviews": {
    "numReviews": "3",
    "reviews": [
      "Great Product!",
      "My favorite shirt!",
      "Highly recommended"
    ],
    "averageStars": "4"
  }
}
```

Direct (Client to Service) PDP Request


PDP Page
http://mysite.com/pdp/123
GET /product/shopper-products/v1/organizations/{orgID}/products/123
GET /inventory?pid=123
GET /reviews?pid=123
Content Delivery Network (CDN)
Salesforce B2C Commerce
Third Party Inventory Service
Third Party Review Service

```
{
  "productDetail": {
    "pid": "123",
    "title": "Red Shirt",
    "descr
a red shir
  }
}
```

```
{
 "inventory": {
  "available": "55",
  "sizes": [
   "s",
   "m",
   "l",
   "xl"
  ]
 }
}
```

```
{
 "reviews": {
  "numReviews": "3",
  "reviews": [
   "Great Product!",
   "My favorite shirt!",
   "Highly recommended"
  ],
  "averageStars": "4"
 }
}
```

NodeJS

A server-side JavaScript runtime built on Chrome's V8 JavaScript engine, NodeJS will be used to build the custom head application(s) that serve API requests where custom business logic is required. There are a variety of libraries available for NodeJS that can be installed into your application using the Node Package Manager (npm). Salesforce also provides a NodeJS SDK (commerce-sdk) that can be used to interact with Salesforce B2C Commerce, which we will cover in more detail in this guide.

MongoDB

A NoSQL document-based distributed database, MongoDB is used in the sample architecture as the database layer for the custom head applications. Consider opting for a managed solution, which will eliminate the need for you to deploy and maintain your own implementation of MongoDB.

Redis

An in-memory, distributed data structure store, Redis is used in the sample architecture as a caching layer for your custom applications, backed by MongoDB. The Salesforce B2C Commerce NodeJS SDK will implement a Redis cache that gives a developer control over caching behavior of SDK calls.

Docker

Docker is a virtualization platform for deploying your custom applications into portable packages called containers. These containers are then deployed and managed by container orchestration software such as Kubernetes.

Kubernetes

Kubernetes (often referred to as "k8s") provides the platform for managing your container workloads. In the sample architecture, Kubernetes manages the custom applications running in Docker containers. Kubernetes is responsible for managing the cluster of your applications, which includes autoscaling, resource monitoring, health checks, load balancing, and updating software with minimal downtime.

Salesforce B2C Commerce

The Salesforce B2C Commerce platform provides the resources and processing for your ecommerce sites, called a storefront. Salesforce B2C Commerce provides HTTP REST APIs or a NodeJS SDK for managing storefront resources, such as baskets, categories, orders, products, recommendations, promotions and pricing, stores, and more.

Third-Party Systems
These include any system that isn't part of the custom head, or Salesforce B2C Commerce – for example, a loyalty service hosted on infrastructure separate from Salesforce B2C Commerce.

Logging/Monitoring/Application Performance Monitoring (APM)
This will consist of several systems that provide critical insights into the health of your overall solution. Aside from having comprehensive logging built into your custom applications, you should also have tools that store and aggregate logs, such as Splunk. Include tools for monitoring and alerting you about the health of critical systems such as Grafana or AppDynamics. We will cover this in more detail.

Now that you have an understanding of the sample architecture and purpose of each component, we will cover some best practices for a successful headless implementation.

3 Optimizing Page Load Time

Aside from having an appealing storefront design, page load time is the most important technical factor in achieving optimal conversion rates. Page load time not only impacts how long users will stay, but it also affects how users reach your site in the first place. Major search engines such as Google use page load time as a factor when ranking your pages in their search results.

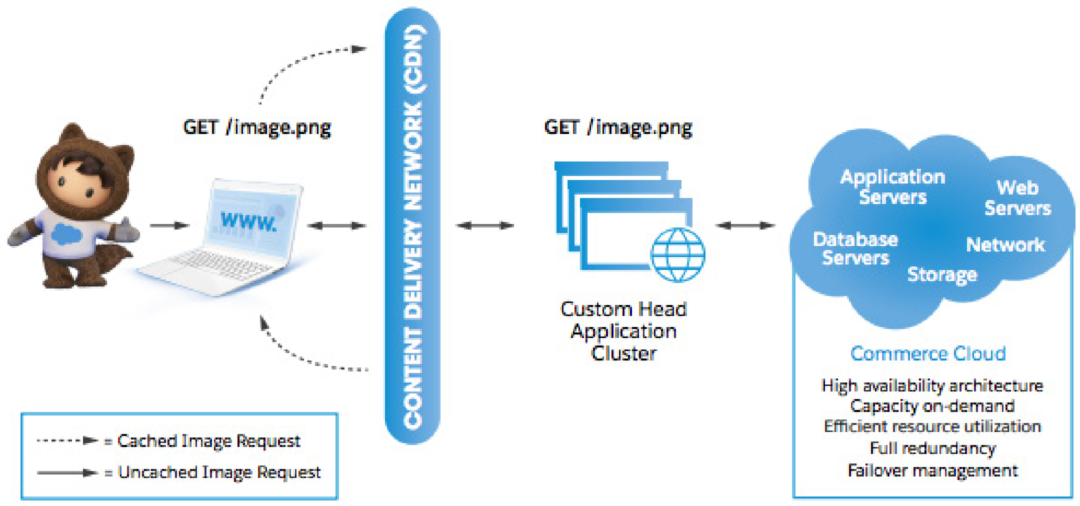
While some of the recommendations below require significant implementation effort, page load time affects your bottom line and therefore these should be considered for any headless implementation.

Content Delivery Network (CDN)

The CDN is typically the first point of entry into your application stack. The more content that can be served by your CDN, the less you need to serve from your infrastructure. This means less processing required from your application stack. Most CDNs also provide performance optimizations that can significantly improve the speed in which content is delivered to the user, as well as rule-based filters to eliminate unwanted traffic to your applications.

Caching Static Content

This is your CDN's most important feature. Most websites load over 50% faster with the CDN serving cached static content such as images, CSS, HTML, and JavaScript, with the CDN serving up to 75% to 90% of all user requests from cache. Without the CDN in place, every request to every file required to load your site will need to be served from within your application stack every time a user loads your website, which can overwhelm your application much sooner.



Regardless of which CDN you use in front of your application, you have full control over which and where resources are cached, and for how long they are cached (commonly referred to as Time to Live, or TTL) before a request to the resource needs to be reissued to your application. This is achieved by using a Cache-Control header, with directives dictating the specific behavior of how the content is cached. Refer to this article from Cloudflare for more detail on implementing the Cache-Control header.

Performance Optimization Features

Most CDNs provide performance optimization features that reduce the size of content a client needs to download before your website is fully loaded in the browser.

For example, code minification condenses your HTML, CSS, and JavaScript into much smaller code without impacting execution or how content is rendered on the page. While this makes your code much lighter weight for transport purposes, debugging minified code becomes difficult, due to how the code is obfuscated. To overcome this, consider using a source map that allows you to debug the original version of the code.

We highly recommend compressing static content (CSS, JS, and HTML) using standard compression methods such as gzip when possible. Doing so will significantly reduce the size of files a client needs to download to render a page on your site.

Most CDNs offer many of these types of performance optimization features that will go a long way in reducing the time it takes a user to load your website. Make sure you understand the potential risks involved with these optimization features. Run functional testing against your website before and after, enabling each one to ensure there are no functional regressions.

Application Caching

Now that you understand the importance of caching static and dynamic content at the CDN, focus on how to implement caching within your application stack to prevent your applications from spending time and resources on requests for data that have already been processed. For example, if you have 100 users browsing product categories on your storefront, there is no need for subsequent requests to travel through your entire application stack to calculate the response for each product category request. A number of caching concepts are available to help you reduce load on your stack.

API Caching

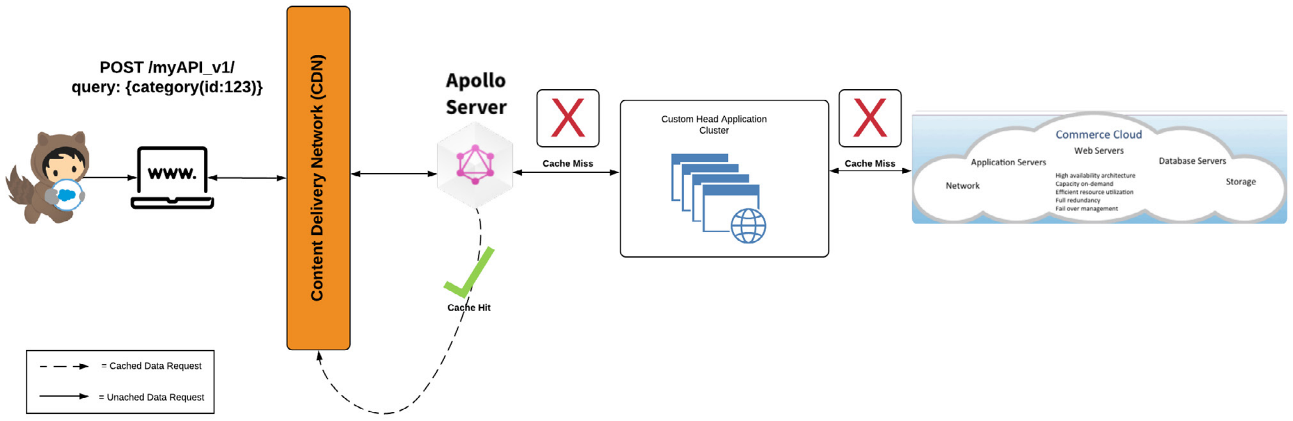
Determining which resources to cache is complicated. It requires thought and effort to ensure users always receive up-to-date and relevant content. Consider grouping your resources to understand which resources are public (eligible for caching), and which are private (will not be cached). Also consider the frequency in which each resource will be changing. For example, product and pricing data may only change once a day, while inventory is constantly changing. We recommend you consider the following exercise prior to implementing caching throughout your application stack:

1. Build a directory of resources based on the endpoints of your application.
2. Determine which resources are public (to be cached) and private (not cached). For example, product pricing would be considered public, while stored payment instruments should be considered private.
3. Determine the frequency in which your data is changing (< 1 minute, >= 10 minutes, >= 1 hour, >= 1 day)
4. Become familiar with caching headers and directives (Cache-Control, max-age, ETag) before implementing caching within your application

With this information, you will be better prepared to implement a caching strategy throughout your application stack.

The Apollo Server can cache and serve cached GraphQL queries to users using the same concepts for the CDN. Since data is often coming from a range of data sources, it's possible that only certain fields involved in the query will be cached. Refer to Caching in Apollo Docs for more detail on the caching flexibility offered by the Apollo server.

Caching at your API layer is well worth the time investment, as it prevents your application from calculating and serving requests that have already been processed for other users.



You should also consider using Apollo's Automatic Persisted Queries (APQ) to take advantage of caching GraphQL queries at your CDN layer, thus reducing network bandwidth and query latency. With APQ enabled, the Apollo client sends the query above as a GET command, and the query response can be served from cache by the CDN.

In-Memory Cache

For requests to your application that are not cached by the CDN or Apollo Server, consider implementing an in-memory cache to allow your application to cache and serve requests that may be expensive to calculate often. In the example of users browsing product categories, consider caching product and pricing data in-memory. This ensures product and pricing requests don't need to be retrieved from Salesforce B2C Commerce APIs every time a user accesses a category page containing product and pricing data.

A number of cache technologies are available for in-memory caching on NodeJS, for example, node-cache. In-memory caches typically store data using key/value pairs, where the key contains a unique string and the value contains the data associated with the key. For example, in a cache containing category results, your key might contain the category ID where the value would contain a JSON representation of a list of products assigned to the category. When your application receives a request to return products in a category, your application can first check the cache to see if the data requested is in the cache. If it isn't, the request will be issued to the back end that contains the product and pricing data (for example, the Salesforce B2C Commerce API). You also can define a maximum number of cached entries allowed in the cache to ensure your cache doesn't consume a significant portion of the memory allocated to your application. Monitoring the size of your cache during load tests helps determine a safe threshold, if one must be imposed.

The in-memory cache is unique for each application node in the cluster. But as users and SEO robots browse your production storefront, the cache will often warm quickly across your application cluster. We recommend allowing the cache-warming process to complete naturally, rather than building cache-warming scripts that may not simulate real traffic.

The B2C Commerce SDK implements an in-memory Redis cache that can be enabled within its client configuration. Refer to SDK Caching for more detail.

Distributed Cache

A distributed cache has the same advantages of an in-memory cache, with a few key differences:

- The cache persists to storage.
- The cache can be shared among application nodes in your cluster. This eliminates the need for cache warming at each application node.

The sample architecture uses Redis for this purpose. A NodeJS npm package is available for it.

Platform API/SDK Cache

The Salesforce B2C Commerce APIs return a `Cache-Control` header, which dictates the TTL for a given API request to be cached. Our NodeJS SDK is capable of implementing a Redis cache which will check for a cached response prior to initiating an API call. By doing so, we reduce the number of API calls made by the SDK, thus improving the performance of applications implementing the SDK. More information about this implementation of Redis within the Commerce NodeJS SDK can be found here.

4 Designing and Planning for Availability

Scaling

As you have learned, appropriate use of caching throughout your infrastructure prevents a high percentage of requests from ever reaching your application cluster. But what about instances where your cache can't protect your application from a higher workload than expected? This can happen in a number of cases:

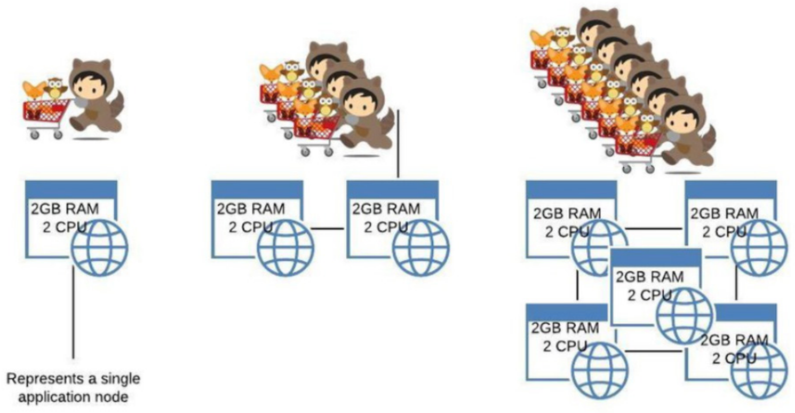
- An increase in requests to endpoints that aren't cached by design (for example, POST requests performing account creations during a big sale event)
- Your cache has been flushed/invalidated
- Your cache is full or incorrectly sized, causing a low cache hit ratio
- You have sent out a marketing campaign containing links with unique parameters that evade your cache. For example, `/myAPI_v1/categories/mens?emailAddress=customer123@email.com`

In these cases, it is important your infrastructure can scale up quickly to handle the increased traffic and avoid performance degradations or service disruptions. Two types of scaling techniques are often implemented in production environments.

Horizontal Scaling

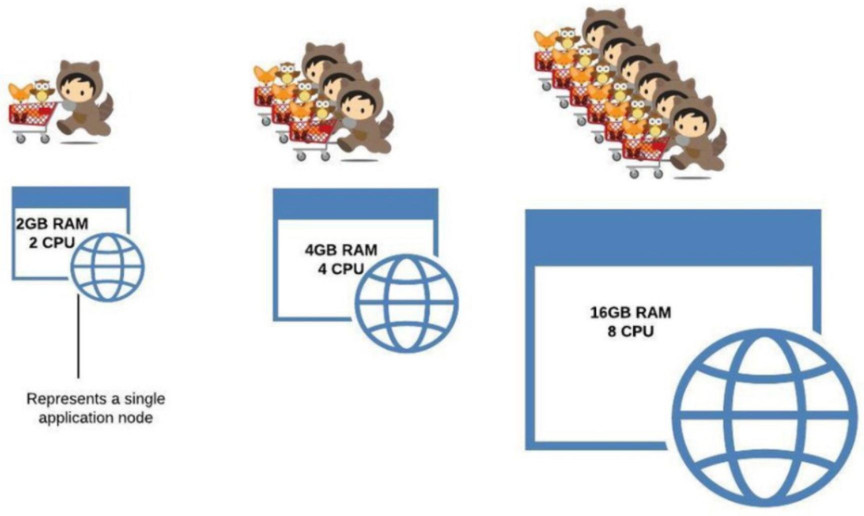
Horizontal scaling allocates more application nodes to a cluster to achieve optimal performance under higher workloads. For example, if there are four application nodes in the cluster with four CPUs and 4GB of RAM, you can simply provision additional application nodes to the cluster, rather than allocate more resources to existing nodes. This is the more common approach in ecommerce, as there are often more limiting factors to supporting higher workloads than just physical resources such as CPU and memory. For example, your application might use a thread pool to manage client tasks with plenty of CPU and memory available before the thread pool becomes exhausted This is where horizontal scaling helps, as adding additional application nodes increases the number of available threads in the pool.

A common misconception about horizontal scaling is that it can be used to absorb infinite load. While horizontal scaling increases the physical system resources available to your applications, doing so during periods of unexpected traffic will put more pressure on other systems (such as your database and third-party endpoints). Understand the constraints of these systems so when your throughput increases, other systems can handle the increased load.



Vertical Scaling

Vertical scaling allocates more physical resources to the application cluster to achieve optimal performance under higher workloads. For example, if there are four application nodes in the cluster with four CPUs and 4GB of RAM, the cluster can be scaled vertically so that the instance is allocated eight CPUs and 16GB of RAM to handle a higher workload. This often requires application nodes be restarted to update the configuration of the cluster. However, this may be an acceptable solution if a single node or fixed sized cluster will handle a higher workload simply by adding more resources. Vertical scaling is common practice for scaling datastore applications such as Redis that might only be deployed as a single node or fixed size cluster.



Kubernetes provides autoscaling features that allow you to scale an application cluster horizontally based on user defined conditions, or even by custom application metrics. For more detail on Kubernetes Autoscaling, refer to Horizontal Pod Autoscaler for more information.

Waiting Room

While your headless implementation is likely to be deployed on a technology stack that includes auto-scaling features, sometimes it's not enough to simply spin up additional application nodes to sustain levels of traffic that are beyond the bounds of what you have tested or expected. Waiting room technologies such as Queue-it integrate with your CDN, allowing you to embed a branded experience into your storefront that controls the level of traffic that is allowed to reach your application, while overflow users are placed in a queue. Over time, implementations mature to handle higher workloads based on lessons learned from load testing and sale events. But in the beginning phases of an implementation where levels of traffic are unknown or higher than what your application currently supports, a waiting room is a good way to safeguard your user experience in the event of sudden performance degradations.

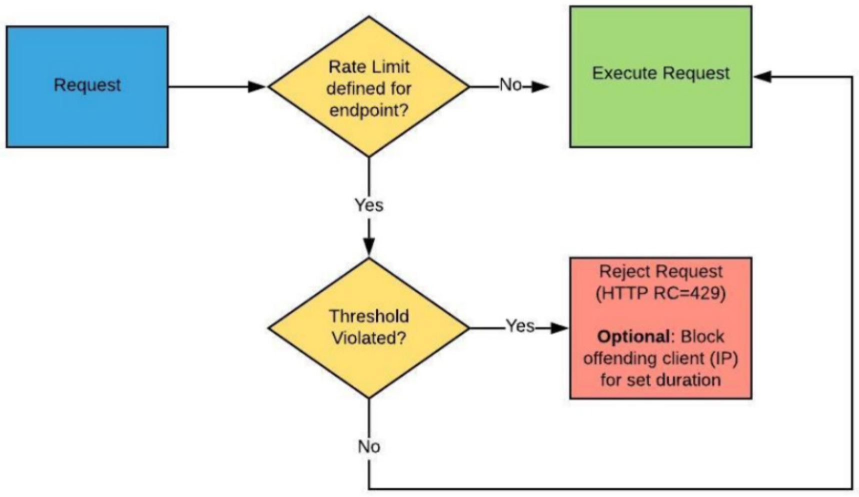
Fault Tolerance

Consider using fault tolerance features such as rate limiting and circuit breaking within your application stack to ensure your system doesn't become overloaded in the event of abuse or sudden bursts in traffic. Through load testing, you will learn the upper limits of each of your APIs and determine at what level performance begins to degrade.

A number of fault tolerance npm packages are available for NodeJS. You can search the npm repository for one that suits your specific needs.

Rate Limiting

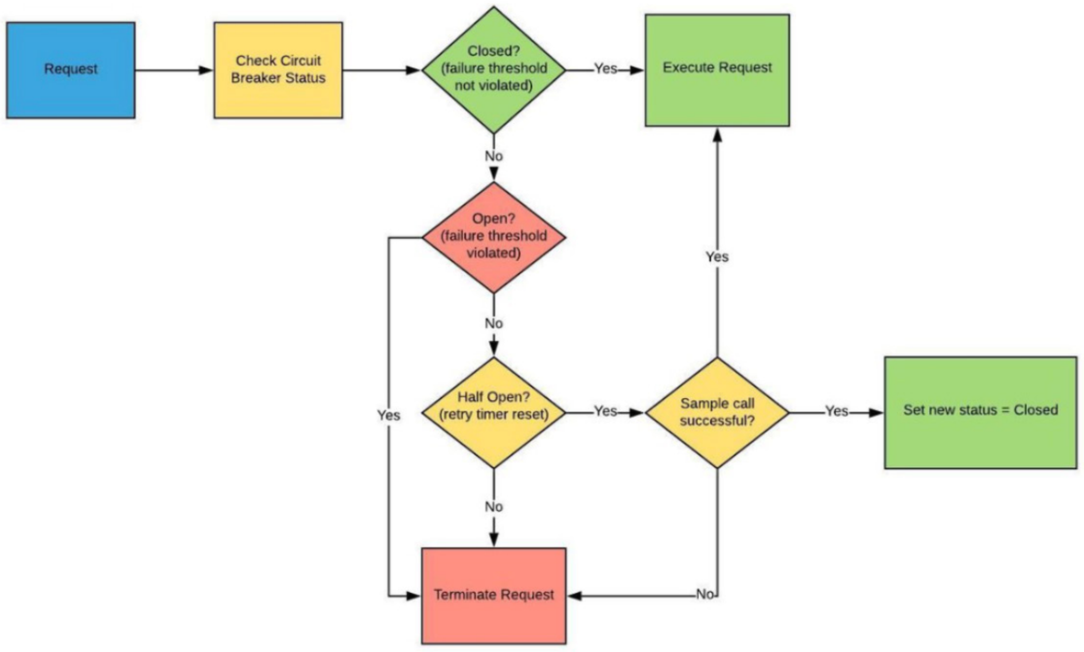
Rate limiting is the concept of defining thresholds for the frequency in which a particular endpoint can be called by a single client or across all clients, and determining what action should be taken if the defined thresholds are violated. For example, if a single client calls your search endpoint more than 100 times in 10 seconds, you might decide to block subsequent requests from that client for 10 minutes. Or you might decide to place the requests in a queue to be processed when the application isn't busy processing other requests from clients who are issuing them at lower rates. The below diagram depicts how requests are handled in a simple rate limiting scenario:


Request
Rate Limit defined for endpoint?
No
Execute Request
Yes
Threshold Violated?
Yes
Reject Request (HTTP RC=429)
Optional: Block offending client (IP) for set duration
No

Circuit Breaking

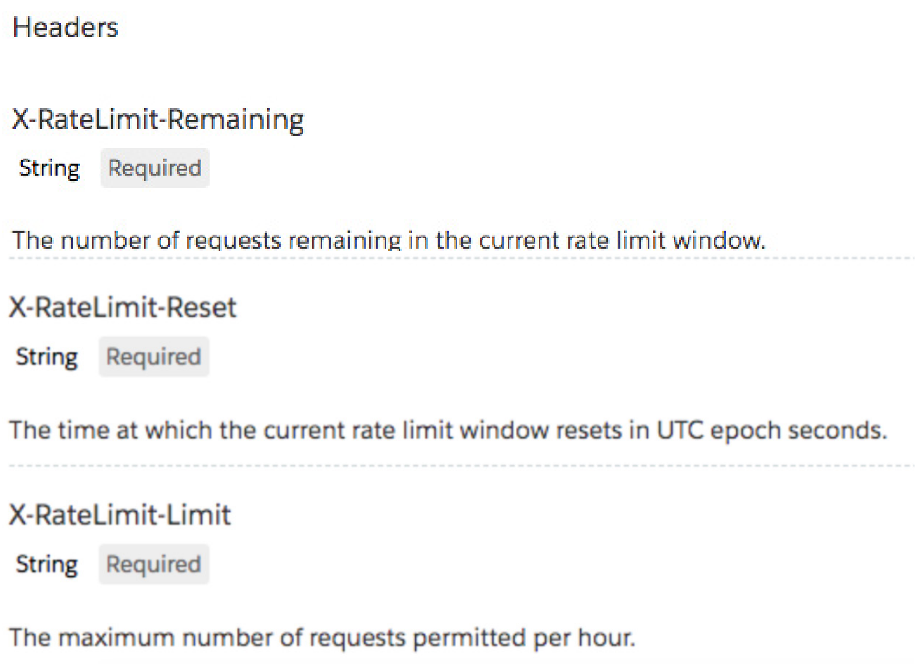
Circuit breaking is the concept of defining upper limits for how long requests to a certain endpoint will remain active before a response is returned, and determining what actions will be taken if the thresholds are violated. For example, if your page load time for logged in customers depends on a call to a third-party loyalty service , you might only want to wait up to two seconds for that call to complete before you close the connection and hide the loyalty point balance element on the page until the user re-loads the page, or an automatic retry is attempted. This is different than simply defining a timeout on an outbound HTTP request, as your application will keep track of overall failures across all users, and open the circuit breaker when certain failure conditions are met within a defined period of time. Doing so prevents your application from spending time attempting requests that are unlikely to be successful.

Below is a diagram illustrating how requests are handled using typical circuit breaker logic:


Request
Check Circuit Breaker Status
Closed? (failure threshold not violated)
Yes
Execute Request
No
Open? (failure threshold violated)
Yes
No
Half Open? (retry timer reset)
Yes
Sample call successful?
Yes
Set new status = Closed
Yes
No
Terminate Request
No

Salesforce B2C Commerce API Fault Tolerance

The Salesforce B2C Commerce APIs have rate limiting capabilities built in, which define the maximum number of requests a tenant is able to issue to a given API per hour. If the number of requests is violated, the client will receive a 429 response code and the request will not be executed. Refer to the API Reference in the Developer Center for documentation on the rate limiting feature. Below is a description of the rate limiting specific headers returned with any Salesforce B2C Commerce API request:



Headers

X-RateLimit-Remaining

String Required

The number of requests remaining in the current rate limit window.

X-RateLimit-Reset

String Required

The time at which the current rate limit window resets in UTC epoch seconds.

X-RateLimit-Limit

String Required

The maximum number of requests permitted per hour.

Load Testing

Load testing is essential to understanding where your application may experience degraded performance, or even unavailability when placed under heavy load. At Salesforce B2C Commerce, we work closely with customers alongside partners who conduct load testing prior to go-live to ensure customer applications and our platform can withstand projected load targets. The following considerations will help you plan for successful load tests:

Preparing for a successful load test

Often times when developing a new system, trying to understand results of a full load test suite can be overwhelming. Prior to running a full test suite, consider testing each component of your application separately when possible. Doing so will help ensure you understand the performance characteristics of each component of your system, including the Salesforce B2C Commerce APIs.

Note: While we recommend load testing Salesforce B2C Commerce APIs, we don't recommend load testing the Einstein Recommendations API, as doing so will negatively impact recommendation results.

When it comes to load testing third-party services, providers often have test endpoints that aren't intended to scale for production level traffic, while testing production level endpoints will incur costs to your organization. Prior to load testing components that leverage third-party APIs, make sure you understand the load in which the service can handle, and also the response time of the service level agreement (SLA) provided by each endpoint. It may be necessary to contact your third-party providers to ensure they allocate appropriate resources for your testing. If using production endpoints for load testing isn't an option, and test endpoints cannot handle your target load, consider mocking the service calls in your code to simulate a realistic delay based on the response time SLA of the third-party service.

Use Realistic Load Scenarios

Forecasting target load can be difficult. As a general rule, test 2x the load of your highest volume sale event from the previous year, or to date. Forecasting the following metrics will help you or your partner design realistic load test scenarios:

- Maximum orders per hour
- Maximum visits/sessions per hour, or page views per hour

It is also important to define a realistic breakdown of the test cases that will be executed by the virtual test users. Doing so will help ensure you aren't stressing endpoints beyond what they will handle in real world scenarios. For example, the number of users in the system checking a gift card balance is substantially lower than the number of users browsing categories and products.

Test Case	% Users Executing	Notes
Homepage	100	
Search	50	User initiates search term from list of popular search terms
Add to cart	65	65% of users will add between 1–10 products to their cart of random quantities between 1–5
Checkout	65	65% of users will complete checkout, using a randomly selected shipping and payment method
Checkout with gift card	5	
Track order	2	
Navigate category	80	User browses 3 random categories and pages through 3 pages of results
Apply coupon	40	40% of users will apply a coupon to their order during checkout
Check gift card balance	1	
Navigate product	85	

Contact Salesforce B2C Commerce Support Prior to Load Testing
It is important to contact Salesforce B2C Commerce support prior to running load tests to ensure the target instance for your test is provisioned with an appropriate initial size capable of handling the target load. This will also prevent sudden increases in traffic from being flagged as potentially malicious.

Contact Salesforce B2C Commerce Client Services for Advanced Load Testing Support
Our client services organization can provide you with an expert who will monitor the Salesforce B2C Commerce platform during your load tests and provide a summary of their findings. This will help pinpoint any potential performance bottlenecks, such as excessive or inefficient API calls that might be contributing to less than optimal response times in your load test report.

Monitor Your Application Stack
Your partner will provide a detailed report highlighting key client-side metrics captured during the load test. But you need to monitor your applications in the event response times are higher than expected. Monitoring high-level system metrics such as CPU and memory utilization of each component will help you understand

how well your custom head scales under load, as well as headroom for handling additional load. We will cover monitoring in more detail.

Load Test Post-Launch

While most customers load test their storefront prior to going live, often customers don't test again, despite significantly changing their storefront design. Load test frequently, especially after introducing major functional changes to help ensure there are no unexpected performance regressions. Consider adding a load test suite to your automated build process, and mark your builds as failed or unstable if performance regressions are observed. Treating performance as a first-class citizen will help you avoid performance regressions as you release new functionality to your storefront.

There are a number of load testing frameworks available for designing and executing your tests, as well as plugins to common build frameworks for trending on performance metrics across builds. To name a few:

- JMeter: An open-source Apache Java application for designing, conducting, and analyzing load tests
- XLT: A free Java-based test automation and load test tool
- OctoPerf: A software-as-a-service (SaaS)-based load testing framework, compatible with JMeter
- Jenkins: A popular open-source build automation engine
    - Performance Plugin: A popular Jenkins plugin that allows you to trend performance baselines across builds and react accordingly by marking your build as good, unstable, or failed

Secure Your Load Test Environment

It's common to isolate non-production environments and applications behind a private network. But in the case of load testing, it's common for a production-like copy of your storefront to be accessible through a custom domain name, such as `loadtest.mystorefront.com`. To ensure your non-production storefront isn't accessible to users or SEO crawlers that could negatively impact your brand, consider the following mechanisms for protecting your storefront:

1. **Whitelist IP ranges** – If your load test environment is behind a CDN, use an IP whitelist configuration to allow access only to authorized IP ranges from which load will be generated
2. **Password Protect** – Implement and enable basic authentication for your application. Enabling basic authentication will require clients to pass valid base64 encoded credentials in an `Authorization` header when issuing requests to your application. For example, if the username and password are `myUser:myPassword`, the load testing client would pass the following header in a request to the server:

   ```
   Authorization: Basic bXlJRDpteVBhc3N3b3Jk
   ```

5 Protecting and Securing the Storefront

Opening your storefront to the public may attract more visitors than you originally anticipated. You should expect friendly and malicious robots to crawl your storefront 24/7, indexing pages to be displayed in search engine results, attempting web-based attacks in attempts to steal sensitive customer data, and more. See how to safeguard your storefront to ensure you are well equipped handle various types of traffic.

Web Application Firewall (WAF)

Having a WAF at the CDN level protects your application stack at the first point of entry by filtering malicious requests that violate rules based on the Open Web Application Security Project (OWASP) Core Rule Set (CRS). These rules are developed based on common web-based attacks often exercised by malicious robots scanning your website for known vulnerabilities. In the case of Cloudflare, the WAF can be enabled in Log-Only Mode for you to review incoming requests that violate OWASP rules before you enable strict (blocking) mode. This practice ensures you are comfortable with the traffic being blocked by your CDN.

Firewall Rules

The Firewall rule configuration provided by the CDN allows you to build conditions and actions to take based on the characteristics of a request being issued to your system. For example, you may want to block access to a public endpoint intended only for employees. To achieve this, you might consider creating a firewall rule that blocks access to the endpoint, unless the request comes from a certain range of IP addresses or contains a certain header in the HTTP request.

Rate Limiting

Many CDNs offer rate limiting capabilities that allow you to get ahead of the inevitable scenarios where malicious robots attempt to gain access to sensitive data such as gift cards, user accounts, or loyalty points. You can set thresholds to limit the amount of times a single IP address can issue requests to certain endpoints. For example, you can create a rate limiting rule that will block a client from your website for one hour if they exceed more than 10 POST requests to your gift card balance check in 10 seconds. While rate limiting isn't a perfect solution to bot mitigation, it helps limit how often malicious bots can attempt to gain access to sensitive data before being blocked. For more sophisticated bot detection and mitigation solutions, consider implementing advanced features of your CDN such as Cloudflare Bot Management, Akamai Bot Manager, as well as partner solutions such as PerimeterX.

Penetration Testing

Penetration testing (often referred to as “pen testing”) is the practice of conducting destructive testing against your system by attempting to exploit a wide range of vulnerabilities, often times web-based, and without any prior knowledge about the systems architecture. The goal is to understand what vulnerabilities exist within the system and determine the impact should these vulnerabilities be exploited in an environment containing sensitive customer data.

While there are tools available for conducting pen tests, we recommend employing a third party with expertise in conducting the tests and presenting the results in a meaningful way. Consider the following when conducting your testing:

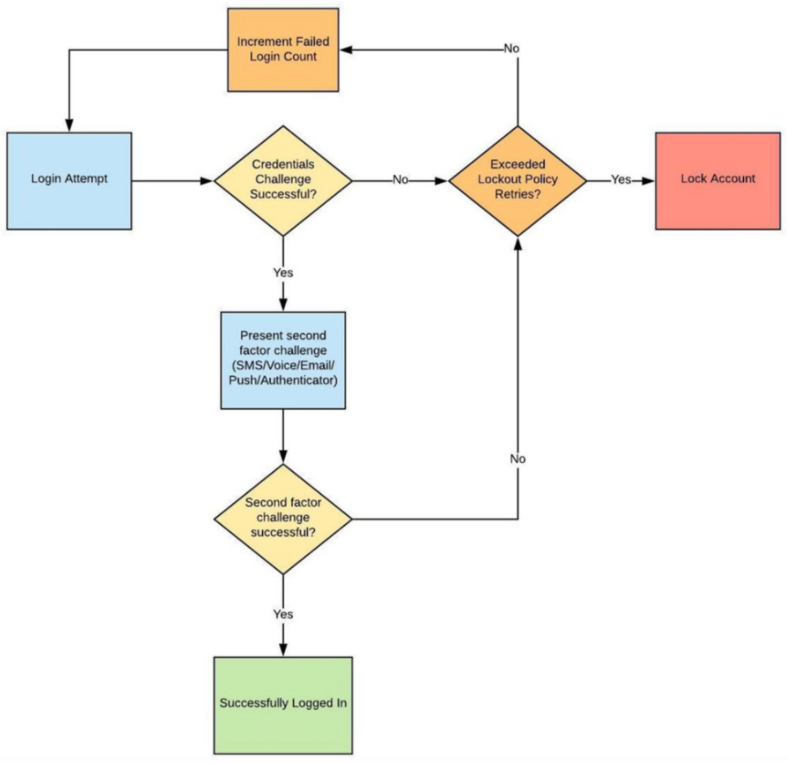
1. Given the destructive nature of pen testing, conduct the testing on an environment that mimics your production environment as closely as possible from an architectural perspective. In the event a pen test is successful in modifying back-end data, you’ll want to make sure you can restore or discard the environment once testing has been completed.
2. Review the results of the pen test with your team and with your third party to make sure you understand which vulnerabilities must be fixed prior to go-live. Have a plan for fixing the lower priority items.
3. Conduct scans regularly, ideally prior to each release. Just because your previous scan was clear of vulnerabilities doesn’t mean that new vulnerabilities haven’t been discovered, or that new versions of libraries in use by your application don’t contain new or undiscovered vulnerabilities.
4. As you review potential vulnerabilities found during testing, consider how your team would have identified the severity of the breach and who might have been affected. We’ll cover logging and monitoring in another section, but think about what data you might need to capture and log to make sure you’re equipped to do a post-mortem analysis in the event of an attempted breach.

Two-Factor Authentication (2FA)

Two-factor authentication (2FA) has become a standard for authenticating to any system. By implementing 2FA within your system, you eliminate the possibility of someone authenticating as another user with stolen credentials alone. After successfully entering a user ID and password, the user must then pass a second authentication challenge that requires access to a secondary account or device that has been registered with your system. For example:

- A verification code provided by email to an address that has been registered with your system
- A verification code provided by SMS to a cell phone that has been registered with your system
- A verification code provided by an automated voice call to a cell phone that has been registered with your system
- A verification code or push notification from an Authenticator application such as Google or Salesforce Authenticator

Below is a simple flow diagram outlining how 2FA works.


Increment Failed Login Count
No
Login Attempt
Credentials Challenge Successful?
No
Exceeded Lockout Policy Retries?
Yes
Lock Account
Yes
Present second factor challenge (SMS/Voice/Email/ Push/Authenticator)
No
Second factor challenge successful?
Yes
Successfully Logged In

2FA should always be implemented for back-end systems such as Salesforce B2C Commerce Business Manager.

Business Manager Access Settings

Aside from enabling 2FA for Salesforce B2C Commerce Business Manager, consider configuring access settings to whitelist or blacklist IP addresses to ensure your Business Manager is only accessible by users on your network.

6 Operational Best Practices

Fine tuning the configuration of middleware components and frameworks that comprise your application stack is important to achieve optimal performance and scale. Monitoring the health of the components that comprise your system during load testing will often help you understand where tuning may be required to remove bottlenecks, but there are some best practices that will help get you started.

Logging and Monitoring

Monitoring and logging is an important aspect of any system, especially one where components are often distributed across disparate networks. Having the ability to monitor the health of your entire application stack in one centralized location can be an extremely powerful tool in monitoring and troubleshooting your system, though this task requires a significant amount of development effort that is well worth the investment. Without real-time and historical visibility into key metrics of your application stack, it's extremely challenging to improve system performance or understand root cause in the event of a service disruption.

Logging and Metrics

The goal of logging throughout your application stack is to be able to quickly troubleshoot any type of problem using nothing but your application logs (no attaching debuggers, no looking at diagnostic dumps, graphs, alerts, etc.). No matter which logging framework you choose, make sure you can log each component of your application in various levels of verbosity, with as little overhead as possible. For example, if the search functionality of your application stops working as expected, know where to find the relevant log entries for a given user session and how to configure log levels for lower level debugging as needed. Make sure you are logging the following types of messages throughout your application stack, whether done through custom logging or as a part of another component of your stack:

- **Correlation IDs** – This is a unique ID used to trace a transaction across all systems involved in your application stack. Consider logging the correlation ID wherever you log a message throughout your application stack, and return the correlation ID as an HTTP response header in your application for troubleshooting purposes.
- **Access logs** – These contain information about each request type (for example, the HTTP request, method, user agent, response size, referrer, and execution time). Many types of web servers log this information by default, but consider adding custom fields such as Correlation ID.
- **Application Server logs** – At a minimum, ERROR level logs from each component of your custom application.

Emitting custom metrics from your applications will significantly help you troubleshoot your application stack. Consider scenarios in which you might need to troubleshoot, then determine which metrics would be helpful in troubleshooting those scenarios. For example, if you anticipate that the response time of your search feature might degrade under certain conditions, consider emitting response time metrics from within your search code so you can monitor the metric in the event of a problem. Whether you find that response time increases with memory consumption or request volume, you will have the insights available for you to react accordingly.

Below are some monitoring tools and frameworks that can help you implement your custom metrics and monitoring:

Prometheus – An open-source framework for storing and monitoring/alerting of time-series data

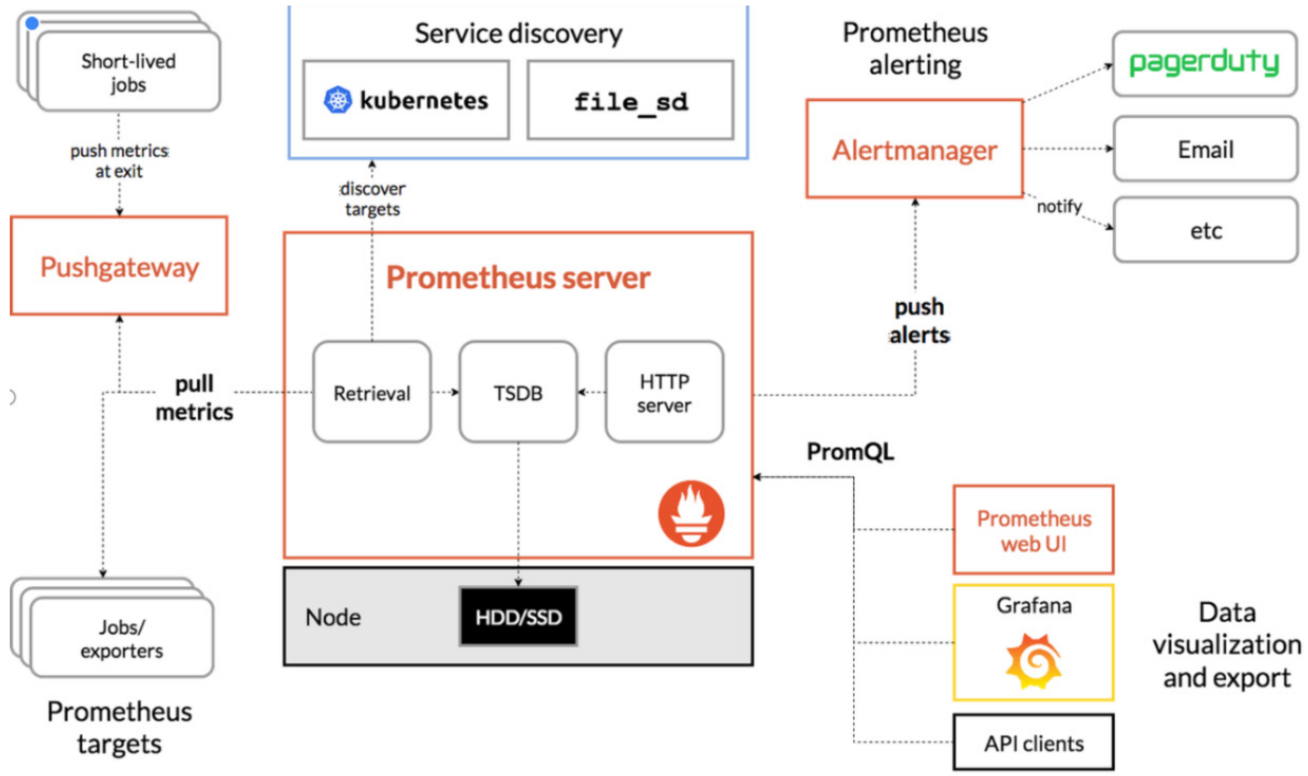
Grafana – An open-source analytics and monitoring solution for a variety of data sources (including Prometheus)

AppDynamics Application Performance Management (APM) – A monitoring and alerting solution for identifying performance bottlenecks, addressing inefficiencies, and improving application performance

Monitoring and Alerting

Once you have defined your logging and metrics strategy, consider how to monitor and alert on the behavior of your system. The framework you choose to implement your monitoring and alerting will dictate the specific features available, but it's most important to alert on metrics that provide you enough time to react prior to service disruptions, whether the reaction is automatic or manual. Refer to Prometheus Alerting Best Practices for an overview of what to alert on and why.

The diagram below depicts the Prometheus monitoring and alerting architecture, along with supported components for visualization and extended alerting:



Prometheus Architecture Diagram by Prometheus Authors is licensed under CC-BY-4.0

You may also consider an all-in-one approach such as Datadog.

Real User Monitoring (RUM)

Real user monitoring (RUM) is the practice of monitoring performance through the lens of users of your application. While monitoring server-side metrics is important to ensure ideal conditions for user experience, RUM allows you to observe what a user experience is like from different locations, browsers, devices, and on different speed networks. This is typically achieved by embedding some JavaScript on certain pages of your storefront, which will compile and send metrics to a RUM framework for gathering valuable insights. This data can then be used to improve important revenue-impacting metrics such as bounce rate, conversion rate, and SEO rankings. You can contact your Salesforce CSM for more information on RUM.

Container Memory Management

Take advantage of Kubernetes memory management by setting reasonable memory limits for your applications to ensure your containers aren't terminated and removed from your application cluster, due to using 100% of allocated memory. Refer to the following articles for more information:

Assign Memory Resources to Containers and Pods

Configure Default Memory Requests and Limits for a Namespace

Running Applications in Production Mode

Some frameworks offer different modes of running an application for debugging or optimized performance. NodeJS Express as well as React both support running an application in development or production modes. Ensure your production applications are set to run in production mode. Refer to the following articles for more information on running in production mode, as well as other helpful tips for optimizing for performance in production environments, such as using React to do code minification:

Express: Production best practices: performance and reliability

React: Optimizing Performance

Client Identification

In a traditional implementation using the Storefront Reference Architecture (SFRA), Salesforce B2C Commerce systems can identify and log the users IP address and their browsers user agent. For example, a request initiated by a Chrome user on Windows will contain a User Agent similar to `Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.117 Safari/537.36.)`.

In a headless implementation, requests executed against the Salesforce B2C Commerce platform will often come from a small set of IP addresses (for example, a handful of CDN load balancers, or an IP address of a custom application issuing requests on behalf of a shopper). In these cases, the requests will often contain generic or default user agents that identify the HTTP client library that initiated the request.

Requests with generic or default user agents (for example, `Python-requests/2.2.1, Axios`, or even an empty user agent) should be avoided in favor of a more descriptive User-Agent, for example, `SalesforceB2C-Checkout-Client_v1.0`. Doing so will make traffic analytics more meaningful, and will also help avoid traffic from your client being flagged as potentially malicious.

While it depends on the HTTP client library you are using to initiate your HTTP requests, setting a user agent can usually be done by simply passing a User-Agent header along with each request which contains a custom value containing the name of your application. As it's common to have a function that prepares an HTTP request by setting static properties such as connection/socket timeouts, adding a property for User-Agent should be a simple step in your implementation.

Continuous Integration and Continuous Deployment (CI/CD)

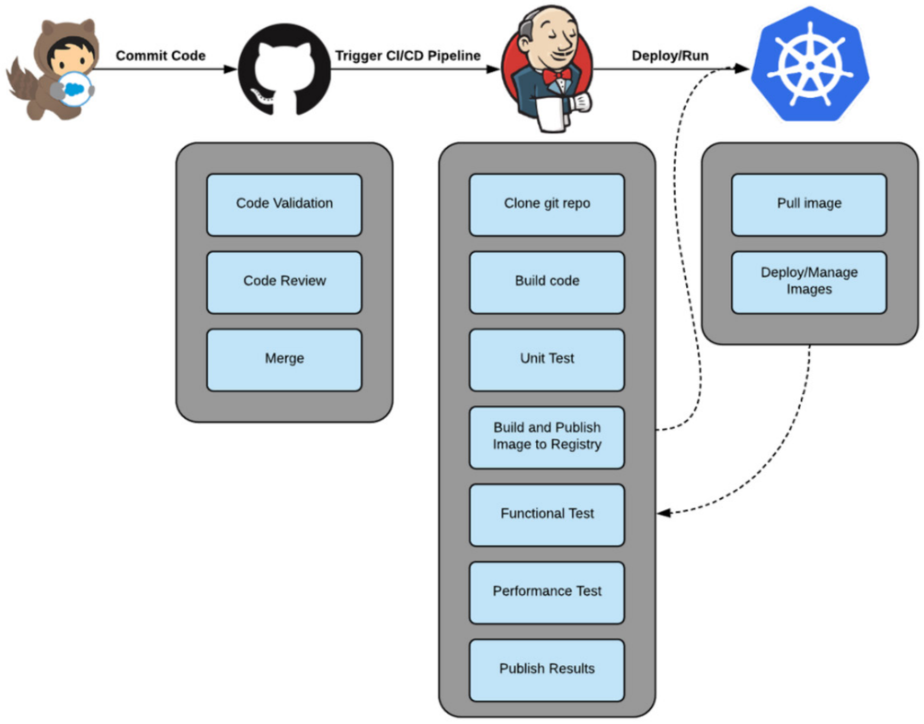
Continuous integration is the practice of managing your code in a central repository (such as Git), and using tooling and automation to provide feedback on the quality of the changes. Through this process, your code is automatically compiled, built, and tested, and the results are published in a central repository where developers can review the status of each phase of the process.

Continuous deployment is the practice of automatically deploying code to mission critical environments such as staging, quality assurance (QA)/dev, and production. This process will also include automatic provisioning of infrastructure in cases where applications are deployed to cloud infrastructure such as AWS.

These practices combined make up your CI/CD pipeline, which has the overall objective of ensuring quality code, free from functional or performance regressions.

While building and maintaining CI/CD pipelines require significant time investment, these processes replace work manually done by developers and operations, usually resulting in human errors. Investing time in building a CI/CD pipeline is highly recommended to save your team time and to help avoid costly mistakes. It's also recommended to trigger alerts on failures within the pipeline to ensure responsible teams can quickly identify the cause and unblock other teams who might be waiting on a new build to be deployed.

The diagram below depicts what a CI/CD pipeline might look like:



7 Leveraging Salesforce B2C Commerce APIs

Now that you have an understanding of the components and practices required for building and maintaining a performant and scalable system, we'll cover how the Salesforce B2C Commerce APIs are accessed from external systems.

REST API

Data stored in Salesforce B2C Commerce can be accessed using a RESTful API. As a developer, you will implement an HTTP client within your application, which will talk directly to the Salesforce B2C Commerce platform over HTTPS. This has been the traditional approach for communicating with Salesforce B2C Commerce prior to the introduction of our NodeJS SDK.

NodeJS SDK

Resources stored in Salesforce B2C Commerce can also be accessed using our NodeJS SDK. If your application is built using NodeJS, we recommend using the SDK to take advantage of built-in best practices, automatic authentication, type-ahead support, error handling, and seamless support for new APIs. The SDK is promise-based, which makes it simple to write asynchronous functions that integrate calls to the SDK.

The SDK is installed by simply running `npm install commerce-sdk`, as described in the SDK documentation.

We will be using Typescript in our examples, which will help us ensure our code is valid prior to execution. To execute the sample SDK code provided below, you can install ts-node or another Typescript compiler of your choosing. `ts-node` makes it simple to compile Typescript into Javascript, and execute it using NodeJS using a single command, while other libraries may require you compile your Typescript code first and run it using a second command.

For additional background on the SDK, refer to the Introducing the Salesforce Commerce SDK article on our engineering blog.

Authentication

Salesforce B2C Commerce requires Authentication (AuthN) and Authorization (AuthZ) for a client to communicate and retrieve data from our APIs. Your application will pass the required authentication token for each API request to ensure your application or user has access to the requested resource. For example, if you're retrieving product data on behalf of a shopper, you need to request a shopper token prior to initiating your API request. Once the token is provided within the API request, Salesforce B2C Commerce will determine if the client is authorized to access the requested resource based on the configuration done by an administrator. If you're using our NodeJS SDK, the authentication is handled for you as part of the method call. Below are the requirements for using Salesforce B2C Commerce APIs:

1. Configure an API Client ID in Account Manager
2. Grant your client access to resources. Instructions can be found on the Developer Center under the **Configure Access** section of the Getting Started page.
3. You will need to know your `Organization ID` and `Short Code`, which are provided by your account executive or customer support manager

4. Identify the security scheme required for issuing your API call. The Developer Center API Reference will tell you which security scheme is required depending on the type of action being performed.

    - ShopperToken – used for accessing resources on behalf of a shopper
    - OAuth 2.0 – used for administrative tasks such as creating, reading, updating, or deleting resources not accessible by shoppers (such as managing coupons, campaigns, and catalog data)

Below is some important information to keep in mind regarding different types of security tokens. Refer to the Commerce Cloud Developer Center for extended documentation on other types of token actions such as logging out, refreshing tokens, or tokens for trusted systems.

Token Type	Token Generation Endpoint	HTTP Method	POST Body	Requirements	Notes	Expiration	SDK Helper Method Available?
Shopper Token	/customers/ actions/login	POST	"type": -{"guest"} {"credentials"} {"session"} {"refresh"}	Path/URL Parameters: • Shortcode • OrganizationId • SiteId • ClientId	• For type=credentials, you must include a base64 encoded user:password in the Authorization header. For example, `Authorization: Basic bXlVc2VybmF tZTpteVBhc3N3b3Jk` • For `type=session`, you must provide valid `dwsid` and `dwsecuretoken` cookies • The Authorization response header from a successful request can be base64 decoded to determine the token expiration date, issue time, token subject (customer id and whether the user is a guest), and token issuer (the client ID)	30 mins	Yes
OAuth2.0 (Account Manager)	https:// account. demandware. com/dwsso/ oauth2/ access_token	POST	grant_ type=client_ credentials	Basic Authorization header containing the client ID and password, encoded in Base64. For example: `Authorization: Basic YWFh YWFh YWFh YWFh YWFh YWFh YWFh YWFh YWFh YWFh OmFh YWFh YWFh YWFh YWFh YWFh YWFh YWFh YWFh YWFh YQ==` `Content-Type: application/x-www-form-urlencoded`	• Sandboxes support client ID and password = `'aaaaaaaaaaaaaaaaaaaaaaaaaaaaa'` • Client permissions must be configured in Business Manager	30 mins	No

Search Example (Guest Shopper)

The example below will walk you through how to perform a product search as a guest shopper using the Salesforce B2C Commerce API using direct HTTP access, as well as the NodeJS SDK. For the HTTP/REST examples, consider using a REST client such as Postman to make it easier to initiate the request and inspect the response body and headers.

Note: The Shopper Token expires 30 minutes from the time of issue. It is recommended you re-use tokens when possible, as doing so will avoid overhead incurred by generating a new token.

HTTP (REST)

1. Visit the Developer Center API Reference which contains detailed documentation for each Salesforce B2C Commerce API Family
2. Filter the `CC API Family` Dropdown list by `Search`
3. Click `View` on the `Shopper Search` tile. Review the `Developer Guide` section for high level details on using the API, which includes the required security scheme (Shopper JWT) for accessing the API.
4. Click on the `API Specification` tab
5. Expand the `Endpoints` section and click on `/product-search`
6. Review the `Overview` section for an understanding of the available methods
7. Click on the method `productSearch`. Review each section for an understanding of required parameters and security requirements. In this case, we are required to provide the following data in a `GET` request to `https://{shortCode}.api.commercecloud.salesforce.com/search/shopper-search/{version}/organizations/{organizationId}/product-search:`
    - A Shopper Token (details on obtaining token are described below)
    - An `organizationId` URI parameter. This ID is customer specific and will be provided to your administrator by support.
    - A `siteId` query parameter. This is the name of the site as defined in Business Manager
    - A `q` query parameter containing the query phrase to search for
8. To retrieve a Shopper Token, initiate a POST request to `https://{shortCode}.api.commercecloud.salesforce.com/customer/shopper-customers/v1/organizations/{organizationId}/customers/actions/login?siteId={siteId}&clientId={clientId}`. The POST request must contain the following:
    - A body containing the following data:

      ```
      {
       "type" : "guest"
      }
      ```
    - Example POST URL:

      POST

      ```
      https://9IUjX4AW.api.commercecloud.salesforce.com/customer/shopper-
      customers/v1/organizations/f_ecom_abcd_stg/customers/actions/
      login?siteId=SiteGenesis&clientId=aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
      ```

9. If the request is successful, the JWT token is returned in a header called `Authorization` with a value `Bearer`
   ```
   eyJfdiI6IjEiLCJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfdiI6IjEiLC
   JleHAiOE1Nzk2N DU0ODcsImlhdCI6MTU3OTY0MzY4NywiaXNzIjoiZjY2ZjBlNGYtZ
   mE0NC00MWViLTliMzU tODlkZTllZTY3ZTcx Iiwic3ViIjoie1wiX3ZcIjpcIjFc
   IixcImN1c3RvbWVyX2luZm9cIjp7XCJjdXN0b21lcl9pZFwiOlwiYWM5SHlhUTcxUDMxOGV
   PT3 JNa21NeUhqdDZcIixcImd1ZXN0X CI6dHJ1ZX19In0.dl8XgldAVo8SGRDrrSA
   dnbD_tnRnfYwIrohjhsPW78JgTif2kukQqnB74RgKHRx6U5CTBee8ktTVwqnmtguRT7y
   iPBTd 2X9f4p14hJKsSE9yjhiLyhnMxRRUGh--2 WANxOFxYnJnf0kl43--pn9FXhbX
   aKSnJCcdLd vbxTuppzLF0fW-yzpLi4caFUidewTrFoFBUj7M4UpF2gd0DtJhX2YEJCTn5jA4y-
   ue4oY7V cp6Y2NpG_mOX_gOmYm_m7hJzKuY4 zU90SGcGYkbBKfRPK3GthTr0LNXVsknydirps
   ZDI1hlBjrCxNz689-ogulmLEncHmjgg3x 84UiPt5_zRst8EmBYcx3H27qXG9_
   HxXixrDevebCBvKsW21gVZk3-RoGsk4N6kLozoG uuA2mzvQ_0qzh_pkQwK02TzcJd2jr6_
   r8PY2Yu83MTOka6GXnpbo2g7jPeP4a8pQYLE VAni6GUniK_zhM1KxaVq3Kw
   ```
10. Now that we have fulfilled the security pre-requisites, we can initiate the API search request with the required parameters:
11. **GET**
    `https://9IUjX4AW.api.commercecloud.salesforce.com/search/shopper-search/v1/organizations/`**`f_ecom_abcd_stg`**`/product-search?count=10&`**`siteId=SiteGenesis&q=suit`**
    - The token must be passed along with the request by including a request header called `Authorization` with the value returned in the `Authorization` header from the previous POST request (`Bearer eyJfdiI6I...`)
    - If successful, you will receive a 200 HTTP Response Code along with the requested data.

SDK

1. Identify the method name you will be using for your API call. In this case, it is the same method identified in step 7 above, `productSearch`.
2. Copy and paste the code from this gist into a file with a `.ts` (typescript) extension. Review the code and comments to understand how the code works.
3. Replace lines 27 - 30 with values for your clientId, organizationId, shortCode, and siteId
4. Execute the code using ts-node <filename>`.ts <search string>`
5. For example: `ts-node productSearch.ts tie`
6. If successful, you will see output similar to below:

```
Your search for tie resulted in 16 hits:
25752235|Checked Silk Tie|USD|29.99
25752986|Striped Silk Tie|USD|29.99
25752981|Striped Silk Tie|USD|29.99
25752218|Solid Silk Tie|USD|29.99
25720817|Tie Front Cardigan|USD|89
25589712|Tie Front Blouse|USD|73.99
25593071|Blouse with Tie Neck|USD|74
```

```
25589059|Tie Front Printed Blouse|USD|80.99
25591144|Dot Tie Front Blouse|USD|47.99
25592386|Floral Tie Front Shirt|USD|64
25545725|Long Sleeve Tie Front Blouse|USD|50.99
25697756|Paisley Sleeveless Shirt With Tie Front.|USD|69
25695461|Raglan Sleeve Tie Front Cardigan.|USD|89
25589408|Tie Front Animal Print Dress|USD|110.99
25688523|Short Sleeve Wrap Blouse with Tie Front|USD|109
25592150|3/4 Button Down Top with Tie Front|USD|69
```

Create a Coupon

Now that you have become familiar navigating the Developer Center, we will skip the steps related to browsing to the Coupon API and describe the steps required for creating a coupon. Since creating a coupon is not a shopper-related function, this API requires an OAuth2 token which is obtained by authenticating to Account Manager. Ensure your client ID is also authorized to create resources of type Coupon, and follow the steps below to obtain an OAuth2 token:

OAuth2 Token Request

1. Execute a `POST` command to https://account.demandware.com/dw/oauth2/access_token with the following headers:

Header Name	Header Value	Notes	Example
Authorization	`Basic <client ID and client password encoded in Base64>`	This is a Base64 encoded value of your API Client ID and password, separated by a colon. For example, 12345:myPassw0rd, encoded in Base64	`Basic MTIzNDU 6bXlQYXNzdzByZA==`
Content-Type	`application/x-www-form-urlencoded`		

2. If successful, the server will return a `200` response code with a response body example shown below. You will use the `token_type` and `access_token` values in the Authorization header for the subsequent request to create the coupon.

```
{
 "access_token": "01e6b0c8-ec29-44de-938e-6c965fee5791",
 "scope": "mail",
 "token_type": "Bearer",
 "expires_in": 1799
}
```

HTTP (REST)

The API Reference in the Developer Center says we can use the `putCouponById` method for creating the coupon using an HTTP `PUT` method:

```
https://{shortCode}.api.commercecloud.salesforce.com/pricing/coupons/{version}/
organizations/{organizationId}/coupons/{couponId}
```

The required **URL parameters** as demonstrated in the URL above are:

URL Parameter Name	Description	Example
organizationId	The ID of your organization, obtained from your administrator	f_ecom_abc_dev
couponId	The ID of the coupon to create	MyCoupon

The required **query parameters** as outlined in the API Reference are:

Query Parameter Name	Description	Example
siteId	The identifer of the site that a request is being made in the context of.	SiteGenesis
type	The type of coupon being created (multiple_codes, single_code, system_codes).	single_code
redemptionLimits.limitPerCode	The defined limit on redemption per coupon code. Null is returned if no limit is defined, which means that each code can be redeemed an unlimited number of times.	0
redemptionLimits. limitPerCustomer	The defined limit on redemption of this coupon per customer. Null is returned if no limit is defined, which means that customers can redeem this coupon an unlimited number of times.	0
redemptionLimits. limitPerTimeFrame	The defined limit on redemption per customer per time-frame. Null is returned if no limit is defined, which means that there is no time-specific redemption limit for customers.	

limitPerTimeFrame.redemptionTimeFrame	The time-frame (in days) of the defined limit on redemption per customer per time-frame. Null is returned if no limit is defined, which means that there is no time-specific redemption limit for customers.	24
limitPerTimeFrame.limit		1
couponId	The ID of the coupon to create	MyCoupon
systemCodesConfig.codePrefix	The prefix defined for coupons of type. If no prefix is defined, or coupon is of type single_code or multiple_codes, null is returned	SG
systemCodesConfig.numberOfCodes	The number of codes that can be generated	50000
enabled	Defines whether the coupon is enabled or disabled	TRUE
description	A description of the coupon being created	This is a sample coupon created using the implementation guide

The required **request headers** are:

Header Name	Value	Example
Authorization	<token_type> <access_token> as returned by the previous OAuth2 request	Bearer 01e6b0c8-ec29-44de-938e-6c965fee5791
Content-Type	application/json	

Based off of our example values provided above, the `PUT` body would look as follows:

```
{
 “couponId”: “MyHTTPCoupon”,
   “description”: “This is a sample coupon created using the implementation guide
(HTTP example).\n”,
 “enabled”: true,
 “exportedCodeCount”: 0,
  “redemptionCount”: 3,
 “redemptionLimits”: {
   “limitPerCode”: 0,
   “limitPerCustomer”: 0,
  “limitPerTimeFrame”: {
     “limit”: 1,
     “redemptionTimeFrame”: 24
   }
 },
 “singleCode”: “MyCode”,
 "systemCodesConfig": {
   "codePrefix": "SG",
   "numberOfCodes": 50000
 },
 "totalCodesCount": 50,
 "type": "single_code"
}
```

Below is an example of the PUT command:

```
https://9IUjX4AW.api.commercecloud.salesforce.com/pricing/coupons/v1/
organizations/f_ecom_abc_dev/coupons/MyHTTPCoupon?siteId=SiteGenesis

And the successful (200) response:
{
 "couponId": "MyHTTPCoupon",
 "creationDate": "2020-01-28T17:54:35.288Z",
 "description": "This is a sample coupon created using the implementation guide
(HTTP example).\n",
 "enabled": true,
 "lastModified": "2020-01-28T17:54:35.320Z",
 "redemptionLimits": {
   "limitPerCode": 0,
   "limitPerCustomer": 0,
   "limitPerTimeFrame": {
     "limit": 1,
     "redemptionTimeFrame": 24
   }
 },
 "singleCode": "MyCode",
 "type": "single_code"
}
```

Note: The API Reference in the CCDC will also show what a successful response looks like for any given API method.

SDK

In this example, we will call the `createCoupon` SDK method to create the coupon:

1. Copy and paste the code found in this gist into a file with a `.ts` (typescript) extension:
2. Execute the code using `ts-node <filename>.ts`
3. If successful, you will see a 200 response code.
4. **Optional:** Write some code using your newly acquired skills to retrieve the coupon you just created!

7 Conclusion

This guide has explained what it takes to build a performant and scalable headless application that integrates with Salesforce B2C Commerce. Head over to the Commerce Cloud Developer Center and learn how to set up, create, deploy, and customize our sample headless application that uses the Salesforce B2C Commerce APIs.