



Atom API

04.09.2020

Munir Syed
Hove

Requirement

The Atom API solution provides a service to return an image based on following request parameters: image name, height, width, image type, watermark and background color.

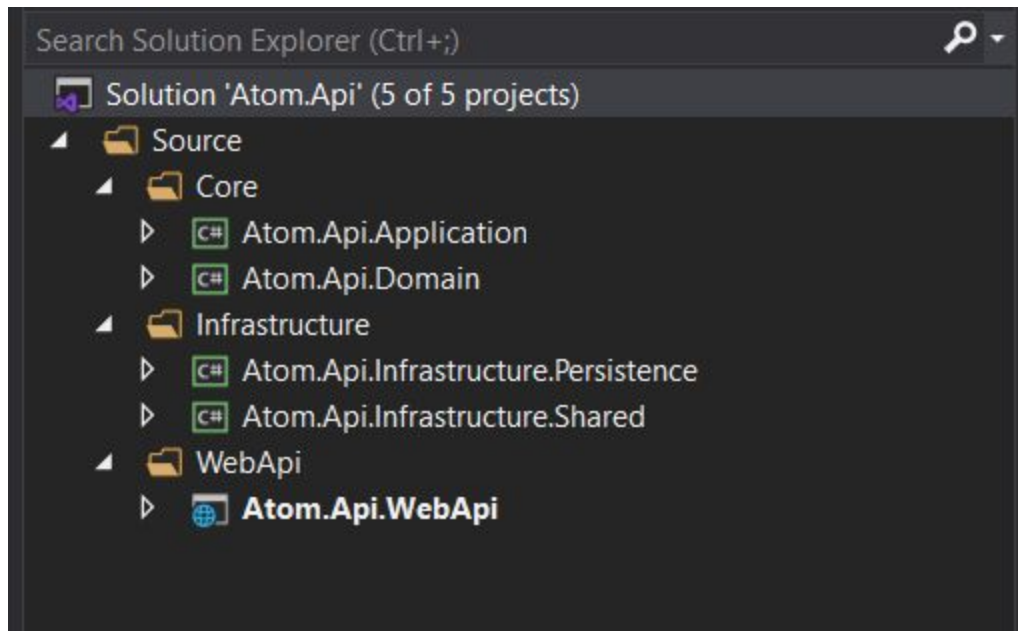
Overview - Patterns and approach used

Clean Architecture

The Atom API solution is built using the clean architecture approach, where layers have been separated out.

With a layered architecture, applications can enforce restrictions on which layers can communicate with other layers. This helps to achieve encapsulation. When a layer is changed or replaced, only those layers that work with it should be impacted. By limiting which layers depend on which other layers, the impact of changes can be mitigated so that a single change doesn't impact the entire application.

The following screenshot shows the folder structures of the solution. The main three layers are Core, Infrastructure and WebApi.

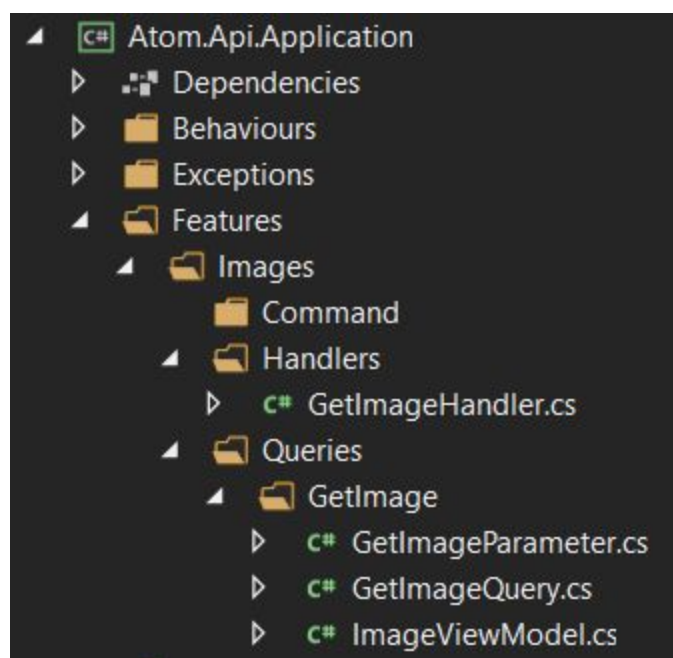


CQRS

CQRS, which is Command Query Responsibility Segregation, is a design pattern that separates the read and write operations and in this design, there is just one query which is to get the file bytes to serve.

However, this implementation enables further enhancements in future with moving into microservices and allows the solution to implement a polyglot persistence architecture with a mix of technologies to achieve the best performance.

The screenshot below shows the CQRS pattern implemented, with Queries folder.



Using MediatR

Using MediatR works well in a CQRS environment as it hides dependencies, thus allows more abstractions on the surface, but with MediatR, it makes decisions on how and when objects interact with each other thus encapsulating the “how” and coordinating execution based on state, the way it is invoked.

MediatR implements the mediator pattern.

With the mediator pattern, communication between objects is encapsulated within a mediator object. Objects no longer communicate directly with each other (decoupling), but instead communicate through the mediator. This reduces the dependencies between communicating objects, thereby reducing coupling.

Another plus point for MediatR is that it can solve the problem of “Constructor Explosion” in API controllers.

Resizing the image and adding text watermarks

Although there are a number of available code examples to resize images, due to time constraints, a tool has been used to do these tasks. It is called LazZiya.ImageResize version 3.0.2

Nuget package: <https://www.nuget.org/packages/LazZiya.ImageResize/>

Scaling the API

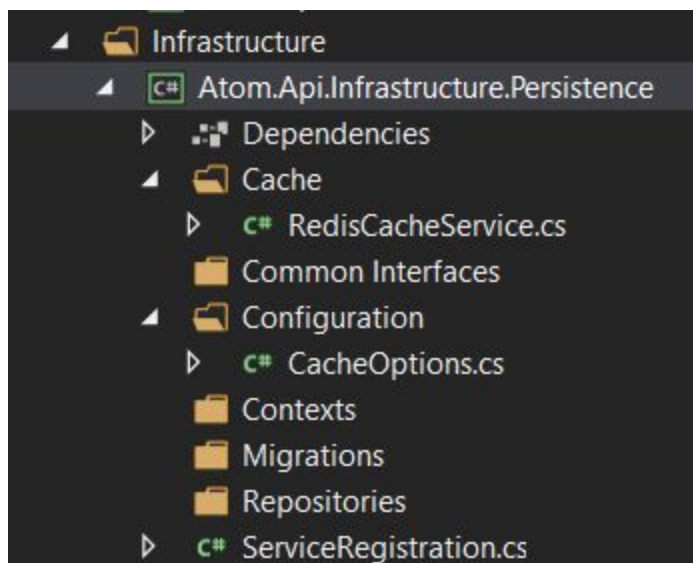
One of the key aspects of serving HTTP requests is improving performance of returning the response object.

In Atom API, the response is always an image served from a local store (for now - may be served from cloud in future?).

Since the response is an image, there is potential to cache the image and served it straight from the cache, if the same image request was made previously and it has been served already.

In order to implement caching in the Infrastructure layer, Redis Cache was used.

As shown in the screenshot below.



Redis Cache - via EasyCaching

Static files, like images, are the right candidate for caching.

Redis is a great choice for implementing a highly available in-memory cache to decrease data access latency, increase throughput and can serve frequently requested items at sub-millisecond response times, and enables you to easily scale for higher loads

Note that the solution uses EasyCaching.Redis library.

EasyCaching is mainly built for .NET Core projects. It contains four basic caching providers: In-Memory, Redis, Memcached, and SQLite.

High availability and scalability

Redis offers a primary-replica architecture in a single node primary or a clustered topology. This allows you to build highly available solutions providing consistent performance and reliability. When there is a need to adjust the cluster size, various options to scale up and scale in or out are also available. This allows the cluster to grow with more demands.

Real-time analytics (to come)

In future, or whenever this is a requirement, Redis can be used with streaming solutions such as Apache Kafka and Amazon Kinesis as an in-memory data store to ingest, process, and analyze real-time data with sub-millisecond latency. Redis is an ideal choice for real-time analytics use cases such as social media analytics, ad targeting, personalization, and IoT

MD5 Hashing algorithm

As Redis Cache is used, and as it's usually the case, the item cached needs a key to uniquely identify it and retrieve it from the store.

The technique used is to concatenate all the input parameters and produce a string that reflects the request and hash this using MD5 algorithm and save this as a key.

To check if the image has already been cached or a better question is to check if the request has been served before with the exact parameters, the same hash technique is used and a key is produced which is then used to check Redis Cache for a previous caching of the request.

This technique is based on how Git works, but not with MD5 algorithm, instead with SHA-1.

HTTP-based Response caching

In addition to the caching of images on the server, a caching directive for browsers is added as a header to the response object, see screenshot below from Network tab, in Developer Tools in Google chrome.



Often the slowest bit of a web API is fetching the data, so, if the data hasn't changed it makes sense to save it in a place that can be retrieved a lot quicker than another API call.

The simplest approach to leveraging standard HTTP caching is to use cache-control.

It also yields the best performance because the client does not need to make any request to the server if the resource is in the client cache

In Atom API, the cache control is set as **public** which means that public platforms such as CDNs can cache, and the age is set to 1800 seconds, or 30 minutes.

Note that the age header sent in the original response is ignored. The middleware computes a new value when serving a cached response.

In the same techniques of improving caching on the client side and saving round trips, the Atom API uses **Etag** in the header. See Etag in the screenshot above.

How is Etag implemented?

In our GET method, the ETag is captured from the If-None-Match HTTP header at the start of the method. At the end of the method, a 304 is returned if the ETag in the If-None-Match HTTP header is the same as from the returned record.

Note with the If-None-Match, the full response is served from cache if the value isn't * and the ETag of the response doesn't match any of the values provided. Otherwise, a 304 (Not Modified) response is served.

Response Compression

ASP.NET Core supports response compression. From popular algorithms, gzip and Brotli are supported.

In Atom API, gzip compression of response is enabled and set to optimal, and the MIME-Type to image/png - however this can be easily extended

Beware, for static files, response compression must be before static files' middleware because otherwise it will not run.

Other approaches to consider for performance benefit:

Other aspects have performance benefits, in mainly speed of delivery, to consider, not yet implement in Atom API but to consider are:

Images on a cloud server, using load balancers (both hardware and software ones) with failover servers, using Anycast networks.

Security

Note in discussing and implementing this solution, Security is slightly left to a side. Things to remember are that DDoS attacks are one of the most substantial threats. Please note that the Atom API currently only supports **HTTP** protocol.

Using the Atom API

Github repository: <https://github.com/Muniro/ImageAPI.git>

Full code available from Git repository at the above address. It can be cloned locally or downloaded as Zip.

Necessary installations

You must install Redis Server as it is needed by the Atom API application.

Download it from <https://github.com/dmajkic/redis/downloads> version 2.4.5 (Uploaded 28 Dec 2011)

A Redis desktop manager can also be downloaded to view Redis in action.

You can download Redis Desktop Manager from <https://redisdesktop.com/>

You can subscribe to a 14 day trial and use it.

Please note ensure you call the local Redis the same name as in the appSettings.json file entry as shown below:

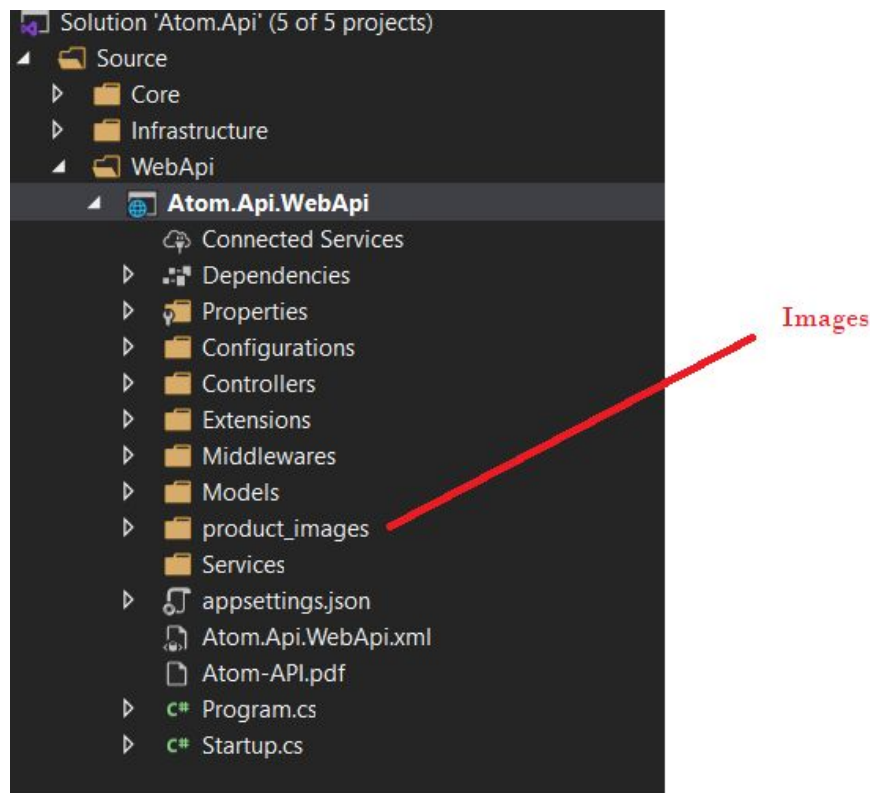
```
"CacheOptions": {  
  "LocalCacheName": "LocalRedis",  
  "CacheAddress": "localhost:6379",  
  "CacheTimeSpanInMinutes": "10"  
},
```

The last key has a value of 10 minutes. That is how long a cached image will live for.

Image folder within the project

Please see where the images are placed. Moving these will require updating the appsettings.json file, as shown below:

```
"ImageOptions": {  
  "FolderName": "product_images"  
},
```

Calling the GET method to get an image

The GET url format takes the shape of:

http://localhost:57712/api/v1/Image/Get/myImage.png/600/500/png/Munir/blue

PATH : api/v1/Image/GET (in blue)

PARAMETERS: (in purple)

myImage.png : the first argument is the image file name

PLEASE NOTE I renamed a file to myImage.png for test purposes. It is still there if you want to use it.

600: is the width

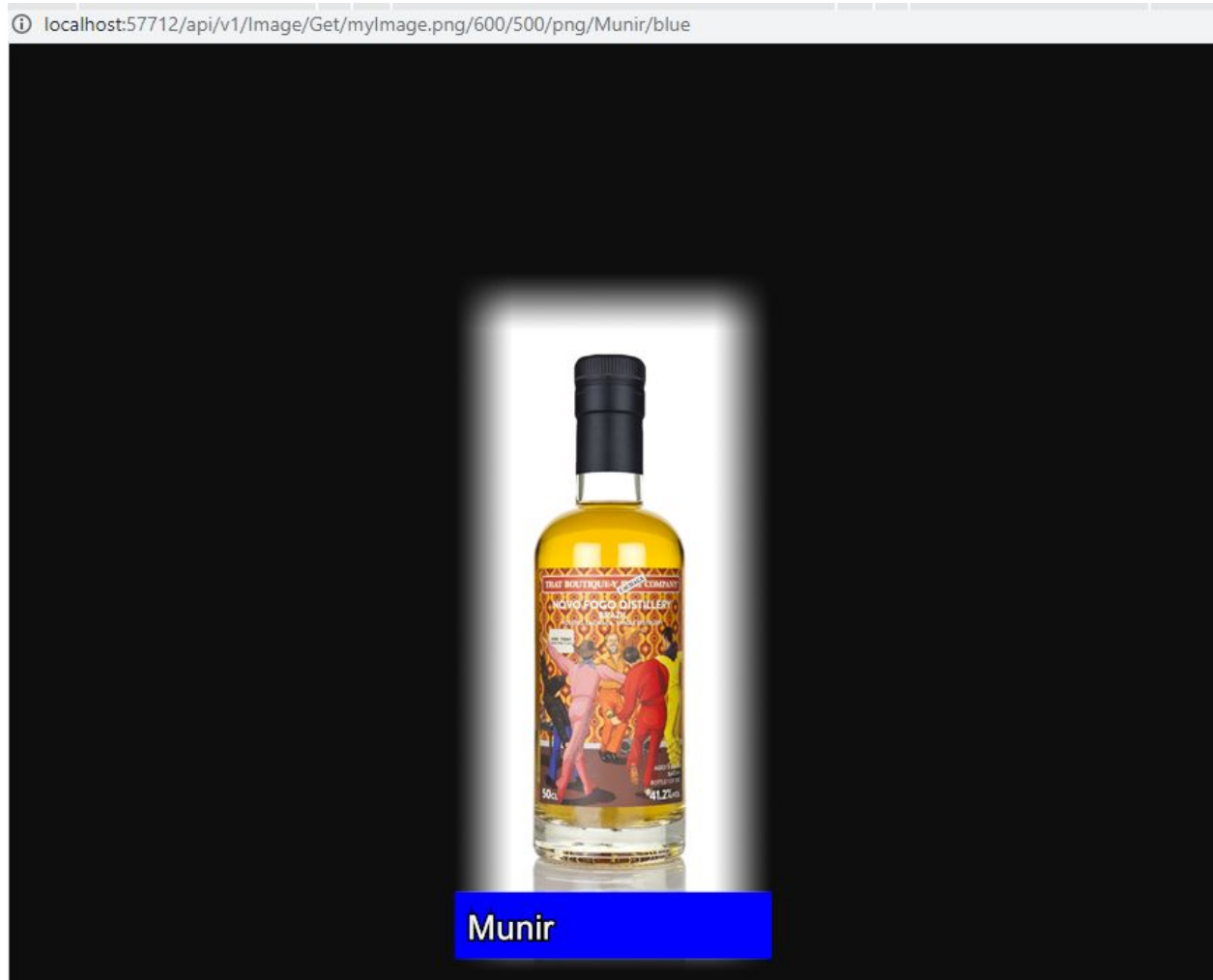
500: is the height

Png:image type

Munir: Watermark text

Blue: background color

Here's a screenshot of the above url after it fetched the image:



The above GET request, running from **POSTMAN**:


GET http://localhost:57712/api/v1/Image/Get/myImage.png/600/500/png/Munir/blue Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (10) Test Results Status: 200 OK Time: 116 ms Size: 122.19 KB Save



Munir

Exceptions

If during the HTTP Get call there is an error, then the system returns this json:

```

1 // 20200904125818
2 // http://localhost:57712/api/v1/Image/Get/myImage.png/600/500/png
3
4 {
5   "Succeeded": false,
6   "Message": "The method or operation is not implemented.",
7   "Errors": null,
8   "Data": null
9 }
```