Caching Strategies and Cache Invalidation

- Handling a large number of courses efficiently
- Planning for cache invalidation

In order to ensure that users of the MicroCourses site experience low latency, a caching strategy will need to be implemented.

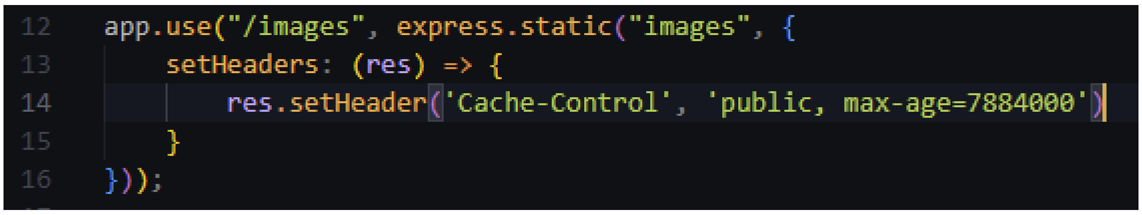
Running under the assumption that course information will only be updated once a semester (and that a semester is defined as three months), the application can take advantage of an aggressive caching strategy with the following techniques:

1. HTTP headers for client side caching

Static content, such as the image files used on the website, can be cached on the user's browser by using the inbuilt express.static() middleware.

Images are already being served via this middleware but are served with a Cache-Control header of 'public, max-age=0'. This means that the response can be stored in a shared cache, but that the cache data expires immediately and the browser will need to ask the server for a new version of the file the next time it wants to display that image.

Assuming this method of serving images is maintained (alternate option explained as part of the discussion around availability), additional arguments can be passed to the middleware to change the way that a user's browser caches images. For example the following code:



```
app.use("/images", express.static("images", {
    setHeaders: (res) => {
        res.setHeader('Cache-Control', 'public, max-age=7884000')
    }
}));
```

instead tells a browser to cache each image for 3 months (7884000 seconds).

As a result of this, the first call to the server for Picture1.png transfers 36kb~ of data in 5ms. The second call instead uses the cached version of the image and doesn't hit the server at all. Though 5ms might seem like a negligible amount of time, this adds up on the course summary page which returns a unique image for each course being offered.

An even more aggressive caching strategy can be utilised by setting the max-age value to 31536000 (one year), adding the 'immutable' argument to the Cache-Control header and assuming that each course's image will have a unique name. The immutable argument tells the browser that the response from the url (the image) will not change and that it doesn't need to do any kind of cache validation, thereby completely removing the need for a request to be made to the server.

While both approaches above the image data would be automatically invalidated after some time, cache data could also be manually invalidated via cache bashing. This involves changing the url at which a file is retrieved, usually by changing the file name. For example the original image might be BackEndWebDevelopment.png, and the replacement would be BackEndWebDevelopmentSept25.png.

The same can be done for css and compiled js files which are also very rarely changed. Node.js automatically creates versioned file names for these files when the application is built using the ‘npm run build’ command.

2. Redis for server side caching

For dynamic content, another method of caching and invalidating data is required. The main overhead in this app is the time it takes for the back-end application to query the db, get the corresponding data and then return that data to the user.

For this the express integration with Redis, a memory based data store, can be used. Though MongoDB provides fast reads via indexing and projection among other things, it still relies on reading from the disk which can be a lengthy process, especially if MongoDB is running on an HDD. Redis on the other hand, being in memory, allows for much more efficient data access.

To implement Redis in the application, all GET API calls would first check the Redis cache to see if the data has been added there. If so it can return that data to the user without querying the DB, if not, the DB will get queried and the result added to the Redis cache so that it can be accessed next time the same data is needed.

In terms of cache invalidation, there are two options. First, automatic validation will be implemented to expire each item in the cache after 1 hour. Though a longer TTL (Time-To-Live, expiration time) value as with the static files could be used, a lower value ensures that rarely accessed pages won’t be held in cache taking up memory for long periods of time. This also means that at most, courses will be out of date for one hour in event that the below manual validation doesn’t apply (courses are updated outside the application).

Secondly, manual cache invalidation can be implemented by expiring data when PUT/POST/DELETE calls are made. For example, if a course is deleted from the database, it can also be removed from the Redis cache. Additionally, if a course is updated, or a new module is added to a course, the same thing can be done to make sure that all users will get the latest data on their next visit to the website.

Redis also has the advantage of being able to scale across multiple nodes and so won’t work counter to the load balancing strategy below. This means that even though a user may hit a different server, they’ll still benefit from the effects of the caching mechanism. More on this below in the discussion around high availability.

Load Balancing

- Distributing incoming requests across multiple instances of an Express server to ensure efficient response times.

Being a globally accessible application, the MicroCourses site must implement load balancing to ensure a good experience for users regardless of the volume or geographical location of the users.

When considering which load balancing algorithm to use, there are several key elements which need to be considered.

1. All content is delivered the same regardless of the user. All users can see all courses and their details.

This means algorithms without server affinity can be utilised.

2. Data is majority text based and will not require long term connections to the server.

Though any single server may handle many requests from multiple users simultaneously there is no reason to believe that any one individual request will consume resources for a substantial period of time. Using the caching strategy outlined above it can be assumed that each request will require a similar, restricted number of resources in terms of processing power.

3. A robust caching strategy has been implemented

Trips to the server and database can be substantially reduced to decrease load on both application and database servers.

For this project I would recommend a round robin algorithm. This allows the distribution of requests across multiple servers without the overhead costs of least connection/least time algorithms. The usual downsides, some servers being overloaded by long lived connections and lack of data persistence between requests are not issues in the application as discussed above.

From there emerges the opportunity to expand this algorithm in two ways as required:

1. Weighted round robin: this would allow for imbalances in processing capacity between individual servers to be overcome. Servers with more powerful hardware would be prioritised over those with consequently decreased processing power.

2. Geolocation based load balancing: if required, and depending on how wide spread the user base is, it may be necessary to implement a hybrid approach which utilizes server pools to decrease location based latency. Traffic would be routed to the

closest server pool and then handled by a round robin algorithm within that server pool.

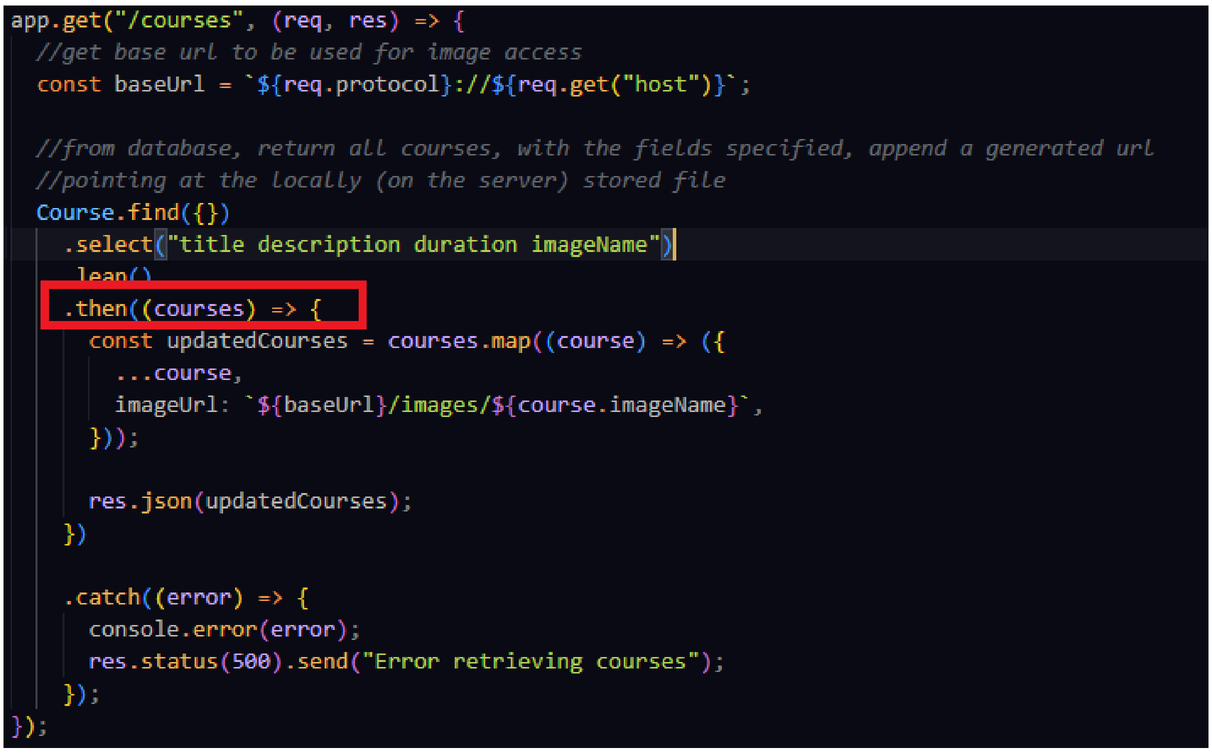
High Availability

- High availability of the application and database in the case of disaster scenarios

High availability of the back-end applications can be ensured in three stages:

1. At the code level

Since NodeJS is a single-threaded framework, it, and by extension Express, utilises asynchronous operations to ensure that any given read/write operation does not block all processing until it is complete. This has already been implemented in the code and should continue to be adhered to as development continues.



```
app.get("/courses", (req, res) => {
  //get base url to be used for image access
  const baseUrl = `${req.protocol}://${req.get("host")}`;

  //from database, return all courses, with the fields specified, append a generated url
  //pointing at the locally (on the server) stored file
  Course.find({})
    .select("title description duration imageName")
    .lean()
    .then((courses) => {
      const updatedCourses = courses.map((course) => ({
        ...course,
        imageUrl: `${baseUrl}/images/${course.imageName}`,
      }));

      res.json(updatedCourses);
    })

    .catch((error) => {
      console.error(error);
      res.status(500).send("Error retrieving courses");
    });
});
```

Basic versions of CI/CD pipelines can be implemented to ensure that no preventable breaking changes are deployed to production environments. These pipelines would ensure that each new code change a) didn't introduce any compilation errors and b) doesn't break any existing functionality. The latter would be achieved by writing tests for all endpoints to ensure that the same input always produces the same output.

2. At the server level

The application should have a mechanism to restart itself in the event of failure, either when the application fails or in the event of a full OS crash. This can be done by process managers such as PM2 as a stop-gap but in the long term docker containers should be utilised.

With regard to process management, docker containers provide for restarting the application the same way PM2 does but has the added benefit of executing health checks to provide information about why the application failed in the first place. This information can then be used to investigate any application side issues.

Containerisation also means that the application can be deployed in the same environmental conditions every time regardless of the device or server that MicroCourses is deployed on. Environmental issues are notoriously difficult to bug, especially if the server is not controlled by the team deploying the application, so less time debugging these kinds of issues leads to more availability overall.

3. At the network level

The application can be accessed by hundreds (or more!) of people around the world simultaneously and so must be able to support that kind of traffic.

The above load balancing strategy is the first way high availability can be ensured at the network level. By having a series of active-active server clusters which can all service any request, each user experiences fast response times for all client-server requests even in the event of increased load.

Similar to the way that testing was implemented in the CI/CD pipelines, testing can also be done on the network as a whole to ensure that the above techniques are working as expected to reduce application downtime. Regular load tests (simulating the maximum expected users) and simulated blackouts should be done to maintain confidence in the system's ability to recover and persist in the event of a real disaster.

High availability in MongoDB:

With regard to the database, MongoDB provides several inbuilt mechanisms to ensure high availability that should be taken advantage of, the most important of which is Replica sets. This functionality allows for the creation of up to 50 copies of the entire database across multiple servers. This means that in the event that one server goes down, another (configured as an automatic failover) can be accessed instead.

Automatic backups of the database should be taken on a regular schedule (daily) to protect against data destruction, tampering and pollution, either via human error or in the case of a security breach by bad faith actors.