Technical Test

Task 1

Question

From what you can observe of this webpage: https://test-test.zptr-p.im/c/prk63utx

- Please provide a technical summary of what you can discern of this www, and breakdown how you would approach building out test coverage of this front end.
- No more than 30-45mins.

Answer

It is a website built with Angular that interacts with a backend api.

Back End tests:

- Unit tests to cover all backend code.
- Feature tests to cover all possible variations of the required backend features.

Front End tests:

- Unit tests to cover all the front end code.
- API tests to check:
  - Correct data returned from the server on each available endpoint.
  - Correct response for invalid requests.
  - Appropriate response from the server when system is down or there is a backend error.
  - All other possible request allowed by the system, like different languages.
- Front End feature tests to:
  - Test each element contains the right text on each page (including pop-ups).
  - Test all provided links in the page work as expected (like “frequently asked questions”, “Find participating stores”, “Tap here for more offers”, etc.).
  - Test all navigation works as expected.

    - Test all action buttons (like “switch retailer”, “I need more assistance”, “Hide/show barcode number”, “Coupon not accepted”, etc.) work as expected.
    - Test other dynamic elements, like countdown or images, work as expected.
    - Test actions that don’t expect user interactions (like when the coupon expires).
    - Test the browser pop-up to know your location works as expected (whether the user clicks yes or no).

For the Front End feature tests, Behat tests can be used to design all the user cases. The drawback of Behat testing is that it can be slow and costly when having a big number of tests. Another option for the feature tests is to use a test runner like Postman, but this way we can only test server responses. Ideally, a combination of both would be acceptable.

Question

In an auto-scaling environment, please discuss potential issues and potential solutions to handle concurrent incoming PUT requests. No more than 30-45mins.

Answer

Assumptions:

- PUT request is used to create or update a resource
- The “entities” handling the requests are web servers since we are dealing with http requests –a PUT request is a type of http request.

We’ll address the problem of concurrent PUT requests from the point of view of the infrastructure and application layers.

From the infrastructure point of view:

A web server is able to handle concurrent requests:

- The web server forks another process to handle the request.
- The process reads the PHP files from disk (several processes can do this at the same time).

However, to avoid slow response times and even system crashes (in case of high traffic) we need to make sure we have a virtual private network where the incoming requests can be evenly distributed by an elastic load balancer. Auto-scaling can then be configured with the help of the load balancer metrics such as the total number of new connections, the total number of concurrent connections, etc. Also, let’s not forget the corresponding grace period when shutting down a web server (this is relevant in the next point).

From the application point of view:

The main problem associated with concurrent PUT requests is data inconsistency: if two or more clients try to update or create the same resource simultaneously, it can lead to the final state of the resource not being what any of the clients intended.

One solution is to use a locking mechanism. Locking ensures that only one client can modify a resource at a time. When a client sends a PUT request, the server acquires a lock on the resource, preventing other clients from modifying it. Once the server has finished processing the request, it releases the lock, allowing other clients to modify the resource. To make this process more efficient, specially for high volume of incoming requests, we can use a cache system to store the status of each lock (Memcached and Redis are two very common cache systems). When using such cache system, it is important to have a grace period for shutting down the servers (auto-scaling) as the cache data will be lost. In case of horizontal scaling we may need to share the lock between instances (i.e. virtual machines) which makes the whole process more complex.

In general, there is no one-size-fits-all approach to implementing a locking mechanism, but rather it depends on the characteristics of the application and is usually implemented on an ad hoc basis. For example, Symfony provides a lock component to create and manage locks (a mechanism to provide exclusive access to shared resources) and it must be implemented in the application for each particular situation.

When updating resources (records) in a database there is a built-in locking mechanism. Depending on the database configuration there may appear deadlock errors which are solved generally by refactoring the functionality.

In the case of asynchronous requests we can use some other approaches like queues or microservices.

- Queues: the intended action of the PUT request can be codified in a message and placed into a FIFO queue (it can also be non FIFO queue). Then, several workers can read these messages, perform the intended action and acknowledge (delete) the message. This still can lead to “deadlock” or “duplicated entry” errors when inserting data in the database for which it is recommended to use a “locking mechanism” (already mentioned). In case of using a locking mechanism, if the resource is lock, the message can just be put back into the queue for another worker to pick it up later.
- Microservices: access system can be delegated and managed in a dedicated service.

Finally, there is the problem of traceability and monitoring when using multiple workers (virtual machines). This problem gets even more complicated when the locks statuses get cached as they need to be shared among workers.