Generic PITFALLS, of exception handling that developers often overlook

```
try {
   my code; my code; my code;
} catch(myException) {

   //console.error ('exception context & app state/variables', exception aka rootCause & stackTrace)
   //in prod-scenario physical persons are not going to look-out for such CLI outputs, need alerts/notifications
   //very tedious to go through console logs 3-4 days old, to zero-in on exact error in multi-user apps
   //generating, logging unique errorIDs and showing them to end-users is basic need, for co-relating user reported errors with logs

   //response.send(exception.httpErrorCode, err message with context and root cause)
   //conveying error to caller may not be sufficient, if caller does not handle properly OR may not be end user using UI
   //many monitoring frameworks may display http response codes but not response bodies

   //logErrorToFileOrDBOrTopicOrElasticSearch( … )
   //ensure persistence till typical resolution time intervals (maybe 3-4 days)
   //ensure traceback of exceptions, using generated unique errorIDs that are shared and screen shot-ed by end users

   //APM.logError( message with context, app state and rootCause)
   //assumes APM server will be accessible & responsive when error / outage occurred
   //cannot be relied upon as sole measure

   //no rethrow OR send response of error will mean code after the catch block is going to execute as-if, no error ever occurred

}
//Code after catch block
```

Reuse is an often, over-quoted word, in software development.
Making a list of the various levels at which software can be <ahem> reused:

1. Reuse at a **conceptual level**, mimic concepts from even other programming languages and frameworks
2. Reuse **source code** snippets, component/s, etc. – copy paste of source code
3. **Compile / runtime reusable code**, packaged as libraries/components, running in-process with main application. Static lib, DLL, node/npm modules
4. Deploy reusable code as **remote services**, which can be used via interfaces / APIs
5. Deploy **remote application** which can provide remote workflows, process flows including human interface UI

User Interfaces (client-side technologies) reuse using Aggregate-able Components

1. code snippets
2. via UI framework (angular, flutter)
3. via raw HTML constructs like div, iframe, etc, including runtime aggregation
4. via specialized applications like portals (liferay, sharepoint)

```
var formBody = {
    "name":"from sender client",
    "mydate": new Date()
}
console.log('mydate being sent as object is');
console.log(formBody.mydate);  //2021-10-14T14:25:32.378Z
console.log('mydate toString is '+formBody.mydate.toLocaleString()); //10/14/2021, 7:55:32 PM

    fetch('http://localhost:3000/', {
        method: 'post',
        body:    JSON.stringify(formBody),
        headers: {
            'Content-Type': 'application/json',
        },
  })
  .then(res1 => res1.json())
  .then(json => {
    console.log(json);
  });
```

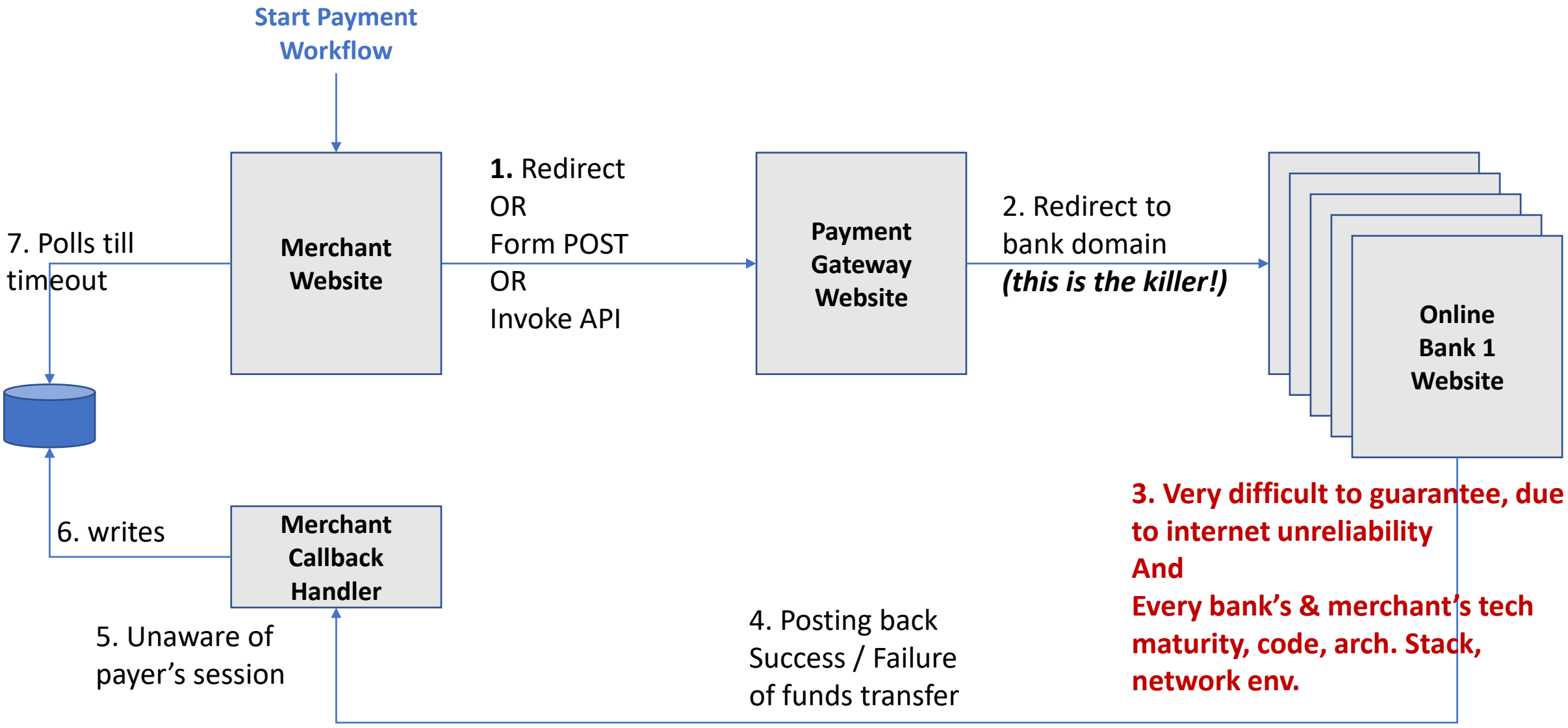
Sending a date object as a POST payload, involves JSON.stringify which implicitly calls the toISOString method on date object and resulting string is sent over the wire.

```
app.post('/', function (req, res) {

  console.log('mydate received as JSON object below:')
  console.log(req.body.mydate.toLocaleString())   //2021-10-14T14:25:32.378Z

  var pdate = new Date(req.body.mydate)
  console.log('mydate reparsed into Date:')
  console.log(pdate.toString())       //Thu Oct 14 2021 19:55:32 GMT+0530 (India Standard Time)

  res.send({"ret":pdate})

})
```

On the receiver side, the default json deserializer cannot ‘retrieve back’ the timezone information.

We will need to ‘kind of reparse’ the date object into a new Date, so that it starts giving out timezone information correctly !

When you pay via internet banking, why is it so flaky?



On web app dashboards:

A Stereotypical screen in a web app, will typically have a get-api in its load and a set-api when a user does a submit action. But dashboards are a whole different story. Some of the worst performing screens in any application are often dashboards, which may also be chosen as a post-login landing page for the web app.

Here are some practical tips while designing dashboard screens:

- Don't show data on dashboards just for tech bravado or to appease egos, how complex is our app's dashboard, damn!
  - **Ask why the user/role needs to see the data**, is she getting insights, are they affecting her subsequent navigation or action? Does the data have business value in the context of the use-case of opening the app?
- **Does the most complex dashboard need to be the post-login, landing page?** Can the full dashboard be separately accessed using an isolated menu item or context menu or link etc.
- Can the dashboard be made to **view data in a nuanced manner**, maybe using tabs, accordion controls or collapse expand page elements. Users will cause the "heavy" API invocation only on a deliberate action rather than the act of login and going to the default landing page
- Different user/roles can have different business purposes, why the same landing page dashboard for all and sundry.
  - Role specific landing pages with user **preference to customize landing page or visible data on the page**
- Show data/stats with **"exception-based management" paradigm**, instead of irrelevant stats of successful transactions
- **High Latency remote API calls from UI**: Numerous APIs (parallel ajax calls) vs a single aggregator API (sync and sequential)
- Pagination and default filters rule, as usual !
- **Default preferences should provide simplest view**, with users wanting increased complexity having to pay the performance price

What is event sourcing

“Event sourcing persists the state of a business entity such an Order or a Customer as a sequence of state-changing events. Whenever the state of a business entity changes, a new event is appended to the list of events. Since saving an event is a single operation, it is inherently atomic.”

When to use event sourcing, a few examples

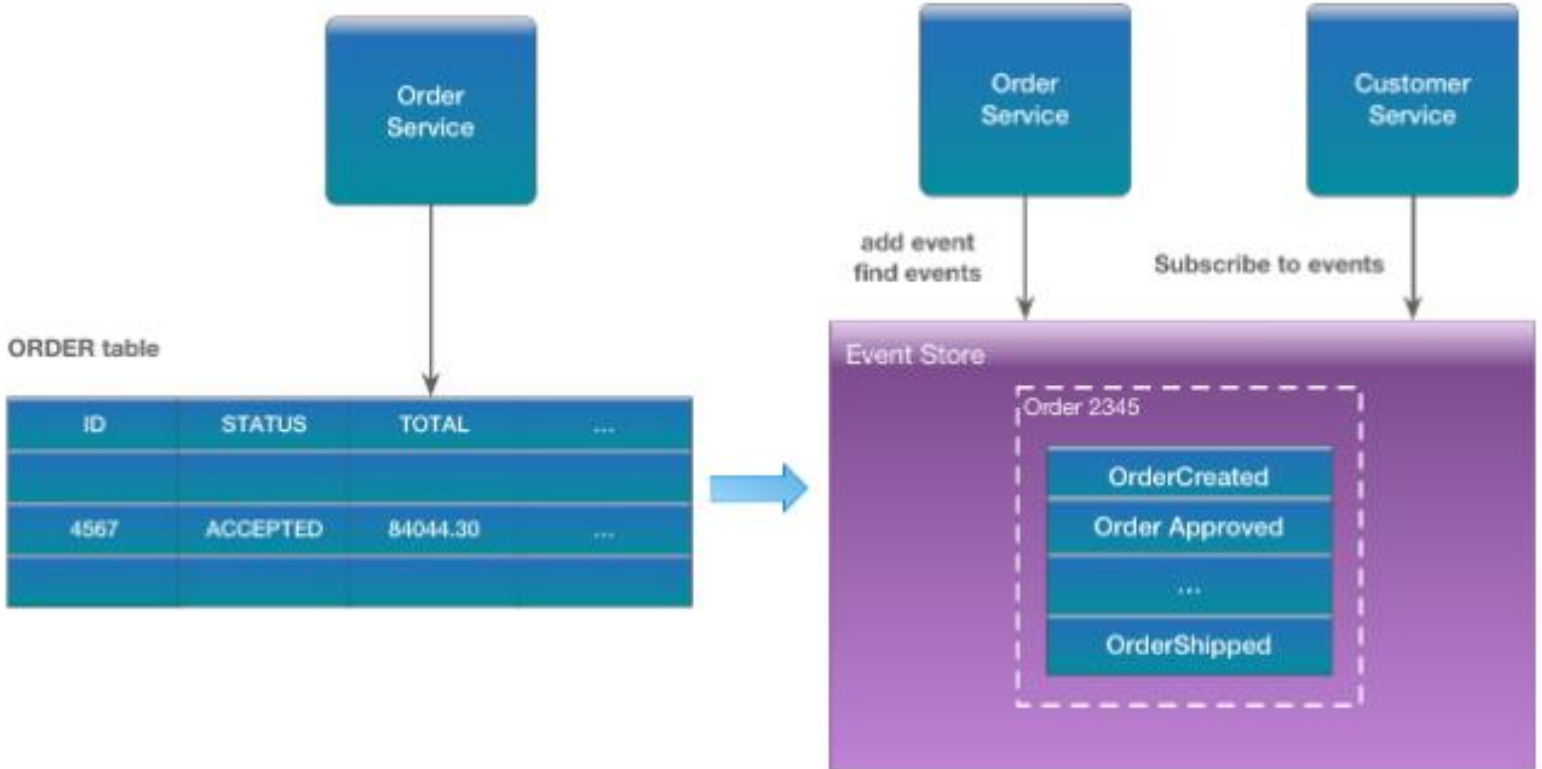
- Change Logs of business entities are mandatory and critical
- Need functional features based on “how” business entity state changed
  - Replay / Undo / redo / reversal of biz transactions
- Need extremely fast inserts, updates, deletes, but relatively infrequent reads/ aggregations, are required
- Accommodation for very large database sizes is easily possible, event sourcing is not denormalized or optimized for storage to say the least

Conventional relational DB:

Insert = append
Update = seek and save record
Delete = seek and delete record
Read = seek and return record

Event Sourcing Data Storage:

Insert = append
Update = append
Delete = append
Read = seek and return record

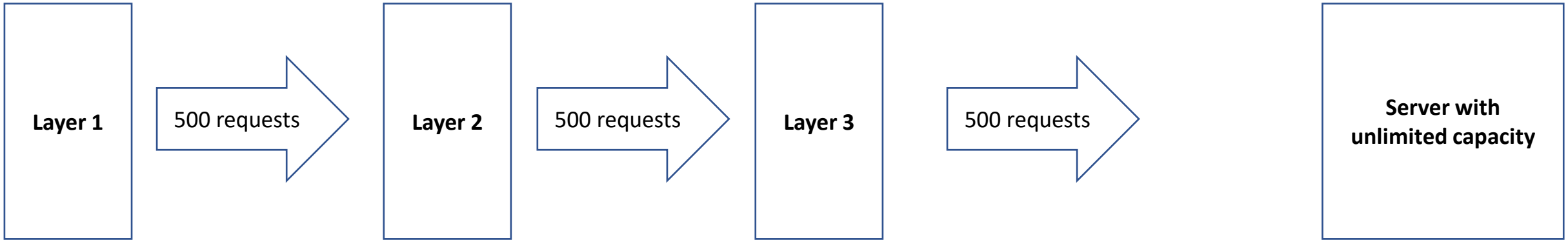


Don’t fall into this trap

1. Select a relational database only because, team is comfortable with it
2. Use a single database across all microservices / bounded contexts
3. End up using classical best practices of relational data model design and result is a highly normalized database
4. Large number of fragmented entities, each with small number of attributes and complex entity relationships
5. To reduce CRUD code in microservices you choose an ORM, whose default APIs may be optimized for writes but not for reads
6. Team finds it easy to quickly expose the CRUD of highly normalized entities as REST APIs
7. Queries are complex and difficult to optimize, due to normalized entities
8. Too many REST calls, per API call from UI, leads to network congestion, increased latency due to numerous remote calls, among point-to-point microservices

Instead

1. Know you domain well, identify bounded contexts and entities within, based on the application’s needs
2. Identify the core entities and the dependent entities accurately for each bounded context, context is king
3. Choose a DB per microservice as per domain’s data model and data access characteristics
4. Denormalization is not all evil and neither is eventual consistency. Remember the CAP theorem. Choose NoSQL aptly.
5. Use Async APIs / messaging to communicate between microservices, where acceptable e.g. Audit Service
6. Minimize remote calls between microservices
7. Introduce caching right from design, it will help you understand your data model better



- All layers can send 500 requests in parallel
- Each request takes 5 seconds processing on the server
- Hence 500 requests are processed in 5 seconds
- Layer 2 sees the ***average request processing time as 5 seconds***

FORMULA DERIVATION:

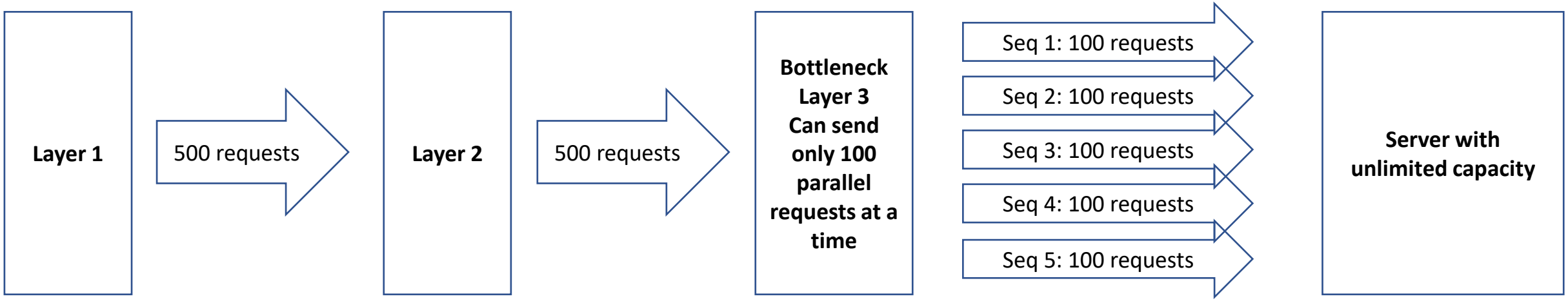
Each request takes R to process in server (in our case R = 5)

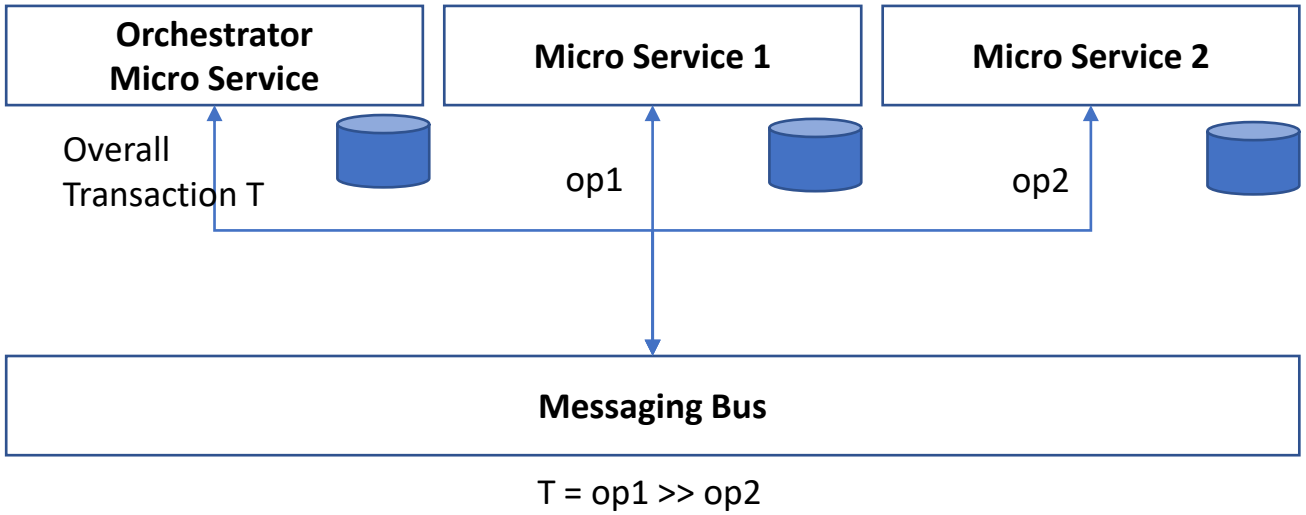
Factor of bottle neck is F (in our case F = 500/100 = 5)

Total time is F + 2F + 3F + 4F + 5F = 15F = F x [N x (N+1) / 2] where N=5

Average processing time = Total Messages / F x [N x (N+1) / 2] where N =5

- **Bottleneck Layer 3 can only send 100 requests in parallel**
- **5 sequences of 100 requests are sent**
- **For layer 2: total time for processing 500 requests is 5 + 10 + 15 +20 + 25 = 75**
- **Hence the *average request processing time now is 6.66 seconds***





T = op1 >> op2

- Async communication between microservices
- Orchestration OR choreography
- Stateless calls
- Local database for each microservice
- Non transactional messaging producers/consumer
- Compensation is an expensive alternative in terms of cost, performance and complexity

1. Orchestrator sends out message m1
2. Consumed by service 1, performs op1, sends ack 1
3. Consumed by orchestrator, moves to next step
4. Orchestrator sends out m2
5. Consumed by service 2, performs op2, sends ack 2
6. Consumed by orchestrator, moves to next step

- The orchestrator needs to provide transactional semantics, by triggering compensating transaction messages OR each service participating in the choreography needs to be capable of transactional semantics
- Also, each service needs to support compensating transactions
- Eventual consistency is best we can hope for

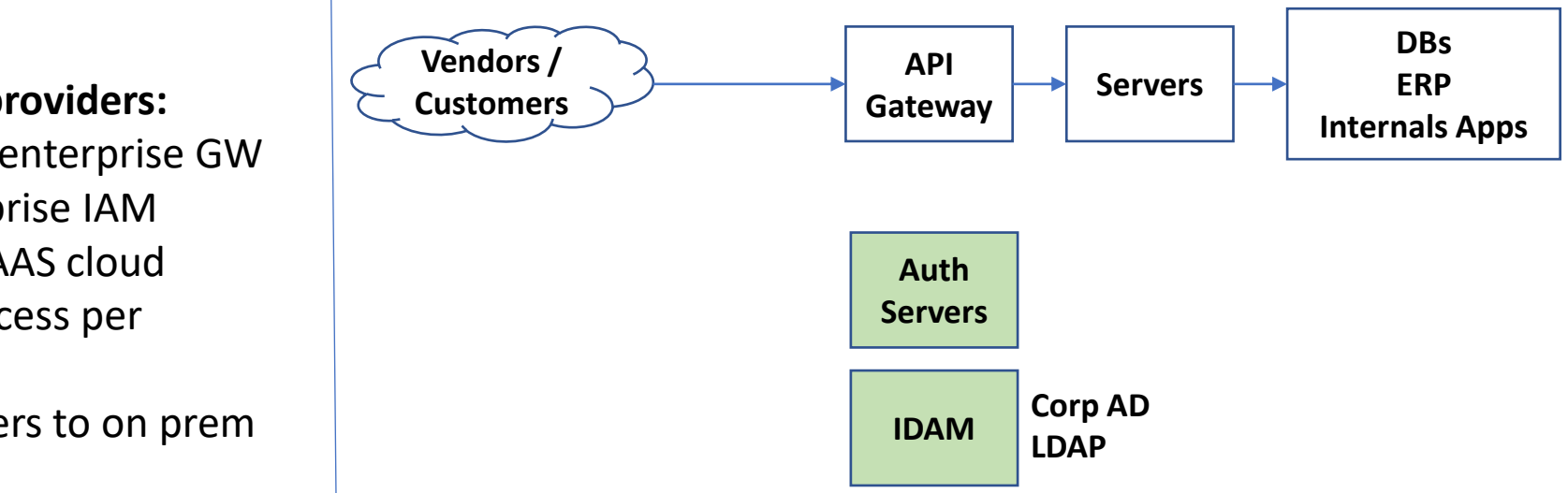
WAF

Cloud based low code development platforms
Eg. PowerApps, Outsystems, Mendix

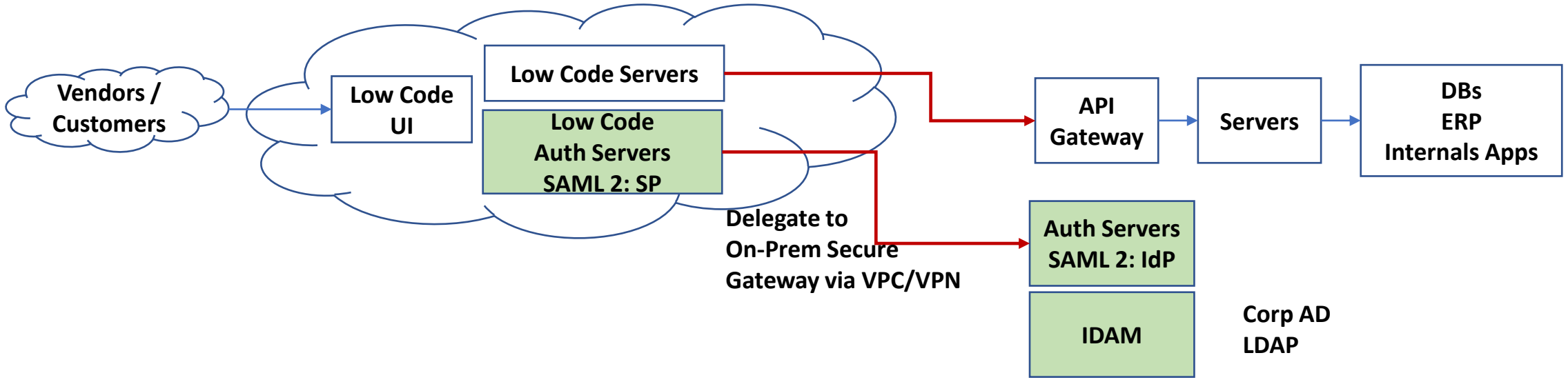
Enterprise Requirements from Low Code providers:

- Secure access from low code servers to enterprise GW
- Secure Auth integrated/SSO with enterprise IAM
- Tenant isolation of access and data in SAAS cloud
  - Dedicated low code servers and access per enterprise
- VPC network extending from SAAS servers to on prem network

Enterprise Apps Deployments



Low Code based Enterprise Apps Deployments

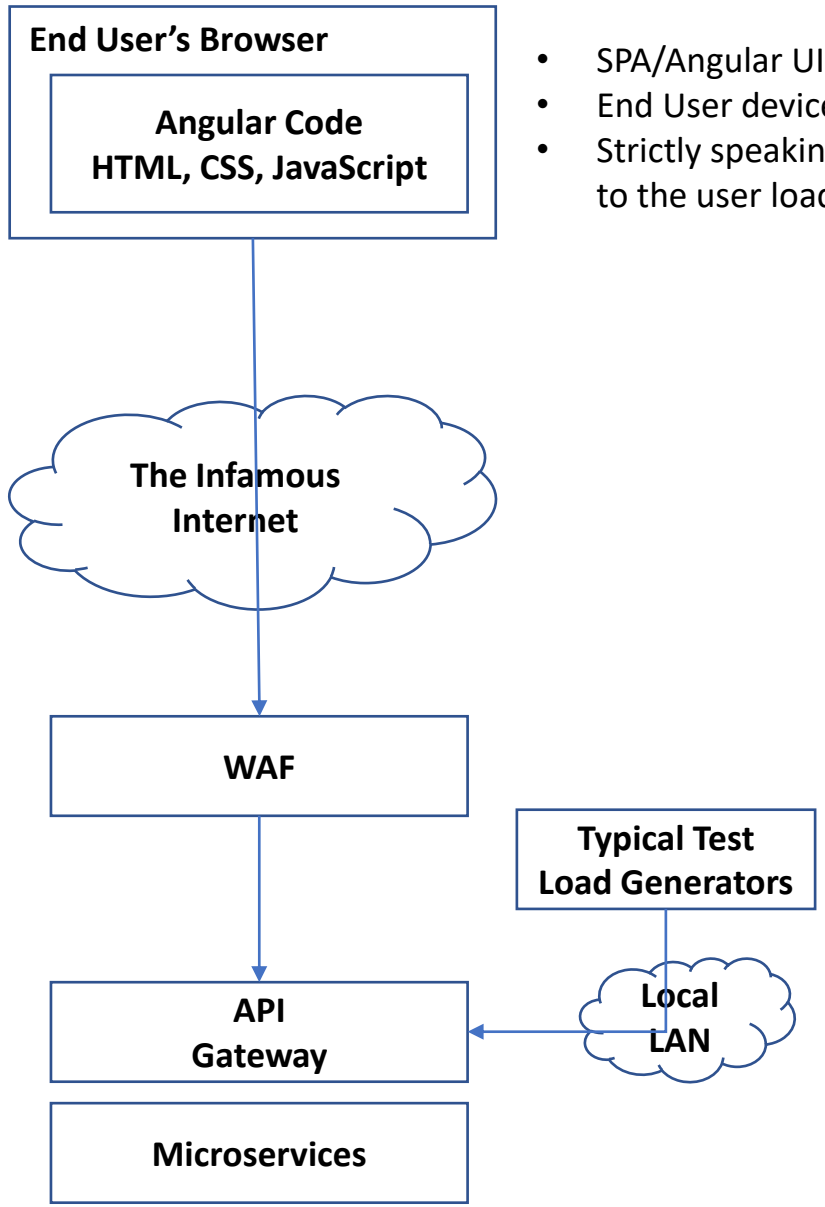


Contrasting load testing VS realistic end-user interactions

Below factors make a real user interaction different from a typical load test

- **User's outdated browsers runtime**
- **Delay in HTML rendering of DOM**
- **Delay in Execution of java script / angular code**
- **Latency of Ajax call/s from the browser**

- **Unpredictable Network Latency**
- **Variable last mile connectivity of end user**



- SPA/Angular UI Execution is on an isolated end user browser
- End User device/machine's CPU, memory is used
- Strictly speaking, UI's own performance is not proportional to the user load

Simulate end user browser behavior

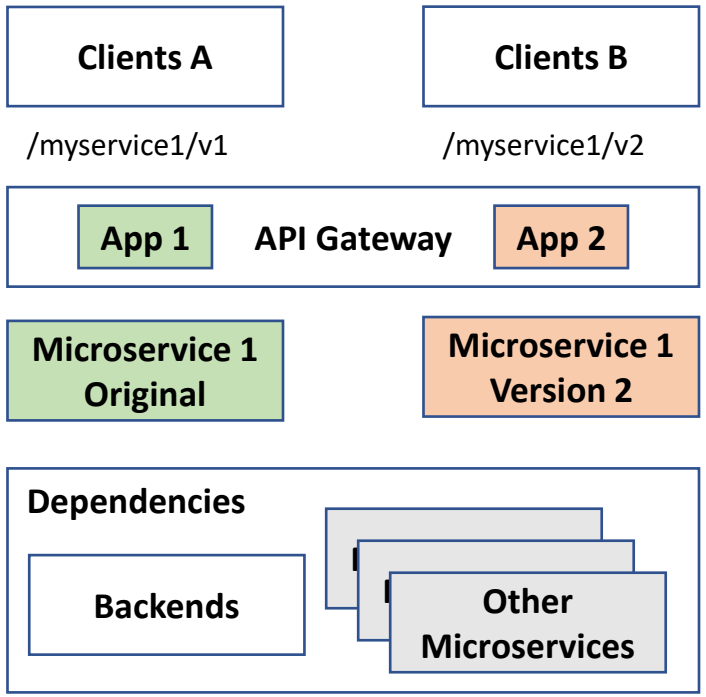
Issue API requests
Parse API responses
Optionally execute limited java script code
Optionally do limited rendering of UI

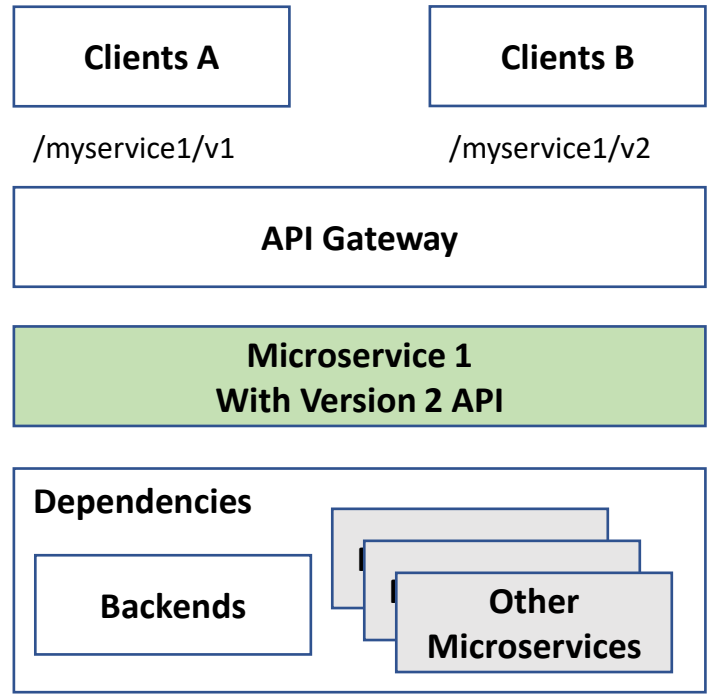
Practical API versioning for microservices

- Approach versioning APIs from a client perspective, never from microservice implementation perspective
- Try and externalize the versioning impact away from your microservices implementation, into infra components like gateways, routers, reverse proxies
- Introducing versioned API on the same microservice can be risky, messy and maintenance nightmare from long term perspective
- Versioning an API at microservice layer may result in API interface changes, code changes, data model changes, backend integration changes, dependent microservices changes etc. Drawing the line between old and new version can be tricky and risky if not designed properly

- Evaluate the **extent of implementation change due to versioning**
- If change is more, a better option is to design the ***new version microservice as a separate deployable unit***
- Use the gateway to redirect traffic to new version microservice based on client specific identifiers

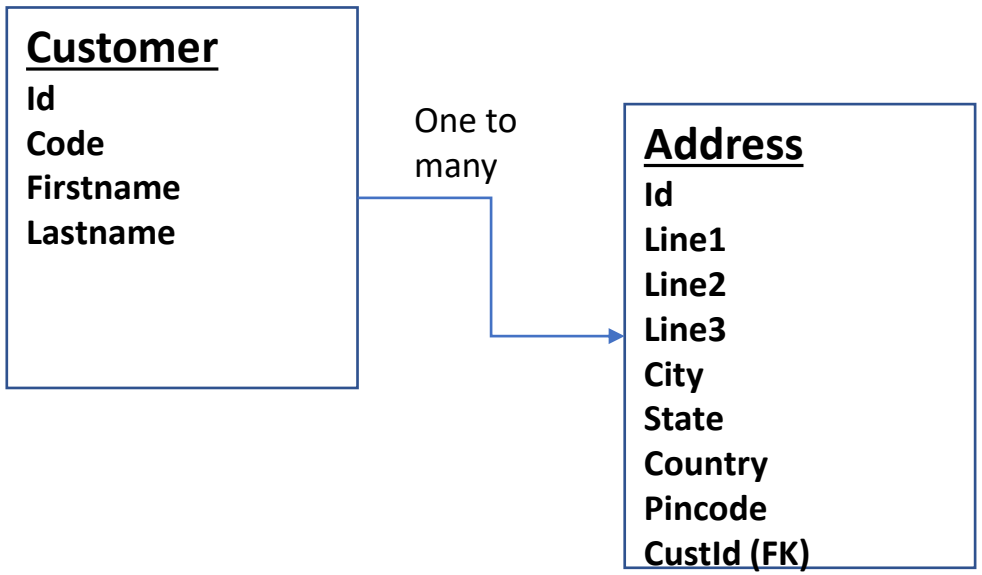
- If change due to versioning is very trivial only then we can think of introducing a new version API on same microservice, at the risk of additional implementation complexity.
- Having same codebase, but different configurations and deployable instances, can also be an option





Transitioning from Relational DB to NoSQL/Document DB

One to Many >> Unidirectional



If “Address” is a dependent entity, then

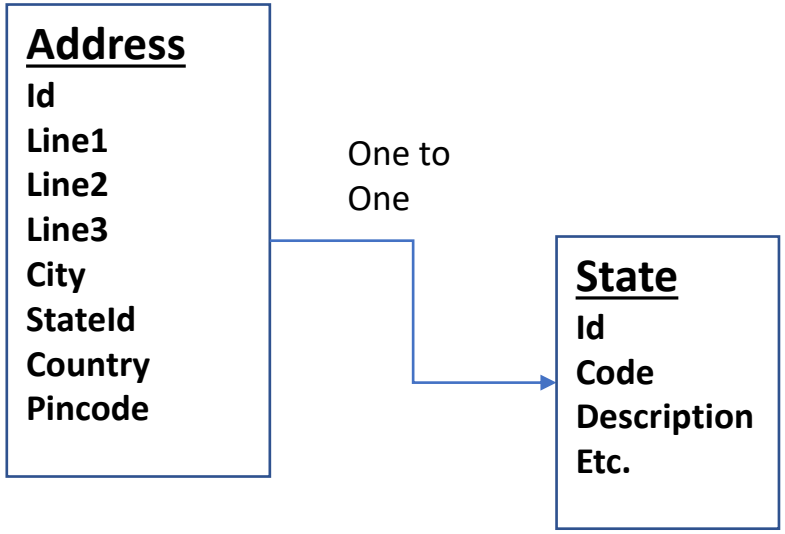
- always the direction of navigation of relationship, will be from the customer to the address
- In the application there is no use case to independently query addresses

The above data relationship can be modelled in mongo as below
Yes it would be possible to query

```
{
    "id": 123,
    "code": "",
    "firstname": "",
    "lastname": "",
    "addresses": [{
        "line1": "",
        "line2": "",
        "line3": "",
        "city": "",
        "state": "",
        "country": "",
        "pincode": ""
    }]
}
```

Transitioning from Relational DB to NoSQL/Document DB

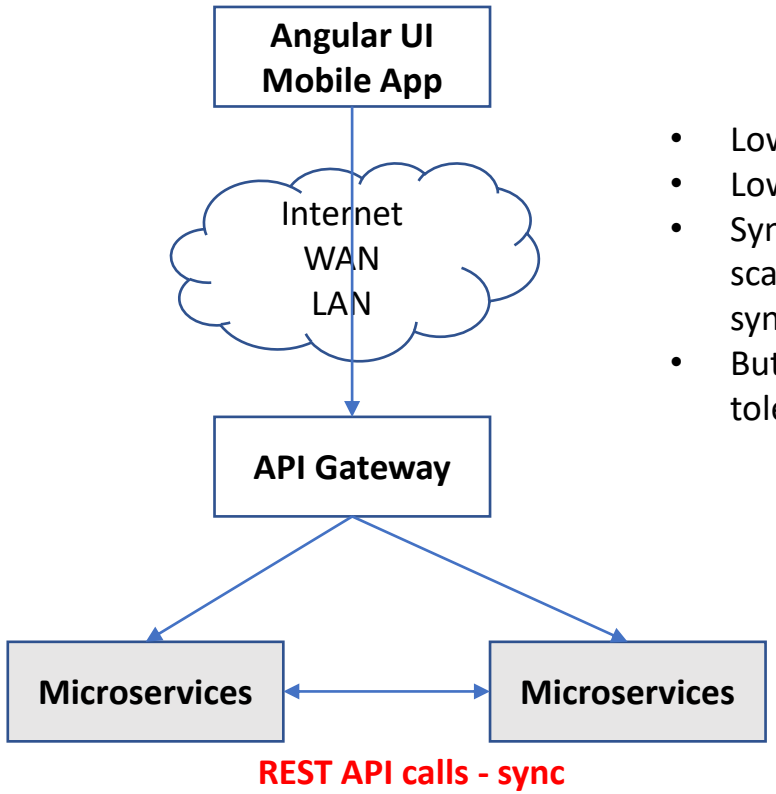
Dealing with Master Data



- Master data keys, in general, should not change/delete, only additions should be possible
- State Code, the business key should never change, the description can change
- If your business requirement is to store master data, as a snapshot, with the transactional data, then you can de-normalize more, by storing the state description along with state code, within the Address data.
- (Retrieval Efficiency = Storage redundancy) V/S consistency
  - description in master changes, it would be impractical to change historical transactional data, if it were a valid biz requirement

```
{
  "id": 123,
  "line1": "",
  "line2": "",
  "line3": "",
  "city":"",
  "state":{
    "code":"MH"
  },
  "country":"",
  "pincode":""

}
```

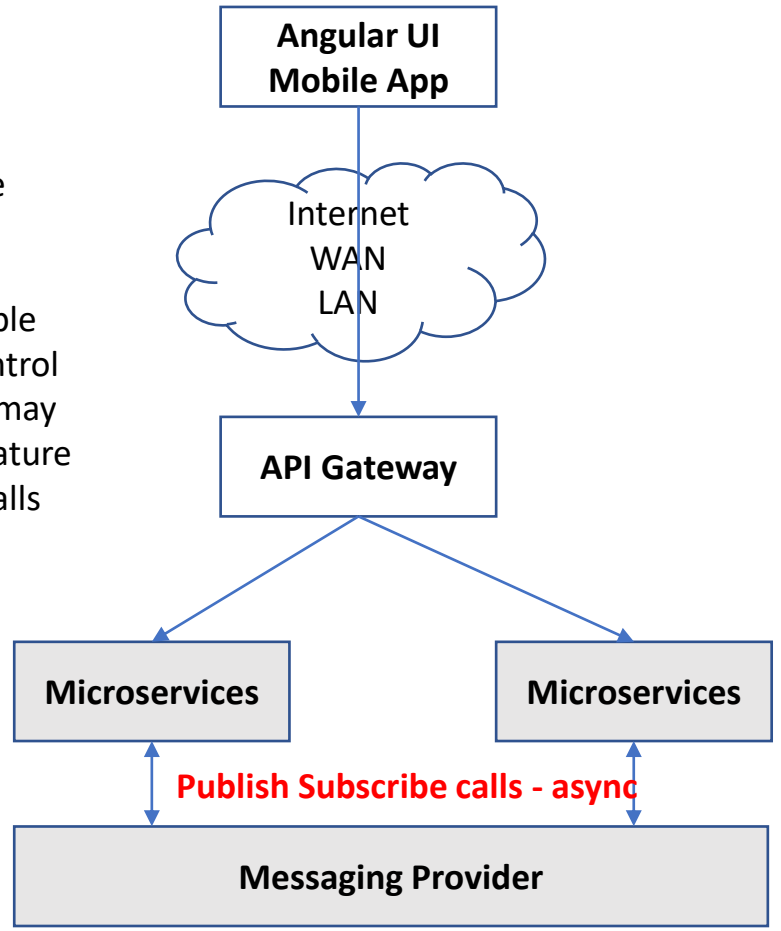
Scalability and Server-Side Latency are often at logger heads with each other



- Low concurrent users
- Low TPM on server side
- Sync REST Calls reduce scalability, due to blocking sync calls inter-microservices
- But server-side latency is tolerable for lower TPMs

Sync REST Calls
Low Latency
Low Scalability

- High concurrent users
- High TPM on server side
- Async Messaging Calls conducive to scalability
- Throttling of TPM possible due to consumption control
- But server-side latency may increase due to async nature of inter-microservices calls



Async Calls
High Latency
High Scalability