Examen SOA1 : Services & Intégration

- Date : 14 novembre 2016
- Durée : 13h30 – 17h30 (4 heures)
- Aucun document autorisé ; barème donné à titre indicatif.

Lisez tout le sujet (6 pages) avant de commencer à répondre aux questions ;
Les questions sont identifiées en gras dans le texte du sujet.
Les différentes parties (au nombre de 3) sont indépendantes.
Pour toutes les questions, la justification de vos réponses est tout aussi importance que votre proposition.

Vous devez répondre aux questions posées dans le sujet sur la copie d'examen fournie, avec d'éventuelles feuilles intercalaires. Vous ne pouvez pas sortir de la salle d'examen durant la première heure (avant 14h30), ni durant le dernier quart d'heure (après 17h15). **Toute fraude identifiée sera systématiquement transmise au conseil de discipline de l'UNS**.

Partie #1 : Prise de recul /5 (1h00)

Dans un article publié en 1968 et intitulé « *GO TO statement considered harmful* », Edgar Dijkstra développait une prise de position très forte à l'encontre des structures de saut « *go to* » pourtant utilisés comme standard de développement dans les langages de l'époque[1]. Depuis, il est d'usage d'intituler « *XXX considered harmful* » des pamphlets (pas toujours de bonne foi) à l'encontre d'une technologie donnée. Le texte de l'annexe 1 est intitulé « *REST considered harmful* », mis en ligne le 24 juillet 2015 par Tomasz Nurkiewicz.

Proposez une analyse critique et synthétique de ce texte (environ 1 page, 300 mots). Justifiez votre analyse, en identifiant dans le texte les arguments acceptables et ceux qui reflètent de la mauvaise foi ou de la non-compréhension du paradigme.

Partie #2 : Services Document /3 (0h30)

Dans ce module, vous avez mis en œuvre des services en suivant trois approches architecturales : RPC, Ressource, et Document. Classiquement, l'approche RPC est développées au-dessus de services SOAP, et l'approche Ressource au-dessus de services REST.

Discutez de manière synthétique (environ 2 paragraphes) l'approche Document pour concevoir un service. Typiquement, comment la mettre en œuvre, quels en sont les avantages et les inconvénients ?

[1] Il proposait l'utilisation de structures de types *loops* et *ifs* pour remplacer les *gotos* qui sont plus expressifs mais n'offrent pas de garantie lors du développement.

Partie #3 : Airport Premier 3000 /12 (2h30)

L'aéroport Nice-Côte d'Azur (NCE) désire mettre en œuvre un nouveau système d'information unifié pour servir de support à son club « Airport Premier ». Ce club est destiné aux passagers réguliers partant de NCE pour les rendre « passagers privilégiés » et ainsi bénéficier de services supplémentaires. L'adhésion au club est gratuite, et chaque membre dispose d'un compte adhérent, ainsi que d'une carte comportant un QR-Code contenant le numéro d'adhérent du porteur.

Les services Airport Premier comprennent (liste non exhaustive) :

- Des lignes dédiées pour passer la sécurité et accéder à la zone d'embarquement, en passant à travers une porte dont l'accès est contrôlé par la carte de membre du club ;
- Des zones réservées dans les parkings « au contact » pour chacun des terminaux (la carte de membre peut ouvrir une barrière à l'intérieur du parking pour accéder à l'étage réservé) ;
- Des réductions (10%) sur les achats dans certaines boutiques de l'aéroport.

Pour bénéficier de ces services, le passager doit tout d'abord être inscrit dans le club, et commencer à cumuler des points CAPS. Il reçoit 100 CAPS en cadeau pour son inscription. Chaque vol pris au départ de NCE rapport 100 CAPS. Le membre déclare son vol en uploadant sa carte d'embarquement électronique depuis le site web (jusqu'à 12 mois), ou en la scannant sur les bornes présentes en salle d'embarquement. On peut aussi cumuler des CAPS en commandant une place de parking en ligne (service Click'n Park) ou en achetant des certaines boutiques de l'aéroport, au taux de 10€ = 1 CAPS. Quand le membre atteint 1000 CAPS, il bénéficie du statut PREMIUM et peut accéder aux services précédemment listés. En atteignant 3000 CAPS, il devient PLATINUM et peut bénéficier de services VIP, comme le Park'n Fly (réduction de 15% sur les parkings, facturation mensuelle et non plus au détail) ou l'accès privilégiés aux salons Club (en utilisant la carte pour accéder au salon, et bénéficier de 20% de réduction). Les CAPS périment 12 mois après l'événement qui a permis de les cumuler.

Le membre dispose d'un site web sur lequel il peut suivre l'évolution de son solde, et télécharger ses cartes d'embarquements. Coté backoffice, on peut suivre l'évolution des populations d'adhérent (membres, premium et platinum), les montants dépensés, identifier les membres allant bientôt basculer de statut pour cibler des campagnes de mailing.

1. **Définissez en pseudo-code les interfaces des services** dont vous avez besoin pour répondre à cette étude de cas. **Justifiez** le (ou les) paradigme(s) de conception de services choisis par rapport à l'étude de cas *(3 points)*
2. Parmi ces services, **quels services sont implémentés par des flots d'intégration** ? Définissez en pseudo-code Camel ou via le langage graphique des EIP l'implémentation de ces flots d'intégration. *(6 points)*
3. Quel rôle joue l'ESB (s'il existe) dans cette architecture ? (*3 points*)

Appendix #1: REST considered harmful

Source: http://www.nurkiewicz.com/2015/07/restful-considered-harmful.html

I don't like RESTful principles and APIs. In recent years it is seen as universal protocol for inter-process communication, especially in distributed systems. However I see many deficiencies of REST and there are alternatives that work well for certain use cases. Obviously there is no one size fits all, I just want to emphasize that REST architecture is flawed in a number of ways.

Bloated, human readable format that demands extra compression

The *de facto* standard format for REST is JSON. At least it's way better than SOAP with its XML and envelopes. Either lots of network bandwidth is wasted (can be an issue with mobile devices or large data centers) or CPU is spent on compression and decompression. A lovely quote:

> *[the] internet is running in debug mode*

It's a serious issue, ask high-frequency traders how hard parsing text-based FIX is. There are plenty of well-established binary protocols that are both easy to parse and use little memory, e.g. Protocol buffers, Avro or Thrift. Moreover backward compatibility is built into them. You can technically be using Google Protocol Buffers with Spring MVC-based REST Services - but somehow very few do it this way and stick to JSON. Of course JSON has one (literally, *one*) tremendous advantage - it's human readable. Well, at least as long as it's pretty printed - rarely the case. But do we really want to pay that price rather than just use binary format by default and have translator or switch when human intervention is required?

Neither schema nor contract

I belong to the static typing camp. I love it when my compiler finds bugs before I even run unit tests - and that I don't need to write so many of them in the first place. Moreover especially in functional programming if code compiles, chances are it works because the contract enforced by types is so strict. When dealing with XML I am happy to have XSD validated so that I can be sure what I read conforms to contract. I can reduce the number of assertions and keep code shorter. Also the XML I produce is guaranteed to be syntactically correct. Last but not least when working with relational databases strict schema prevents me from corrupting data by inserting incorrect values. Similarly to XML, I can also trust what I read: foreign keys are correct, NOT NULL columns really are not null, etc. These are clearly the advantages of machine enforced contract. Somehow all of this is not important anymore when designing API to be produced and consumed by machines in a distributed system (and accidentally in many schema-less NoSQL databases like MongoDB).

Understandable hate of SOAP pushed us to another extreme - no contract at all. There is no widespread standard for documenting contracts and enforcing them automatically for REST APIs. Most of the APIs I find these days are just a collection of sample requests and responses, Swagger at best. I never know which fields are optional, what are the formats and constraints. One might argue that this is by design, that we don't couple ourselves to one concrete API, allowing it evolve. But I have a feeling that most of the

API consumers are nevertheless coupled and will break once undocumented and unannounced change is rolled out.

Publishing URIs

Talking about documentation, purists claim that the only piece of information API should expose is the root URI, e.g. www.example.com/api. Everything else, including allowed methods, resources, content types and documents should be discovered via HATEOAS. And if URIs are not officially documented but instead should be discovered, it implies we should not rely on hard-coded URIs in our code but instead traverse the resource tree every time we use such API. This is idiomatic, however also slow and error-prone. Not to mention Swagger, the *de facto* standard for REST documentation, officially claims such approach is "not per design" - and sticks to fixed URIs.

No support for batching, paging, sorting/searching, ...

RESTful web service does not natively support many enterprise-grade features of APIs like batching requests, paging, sorting, searching and many others. There are competing suggestions, like query parameters, request headers, etc. I remember an hour long discussion about flexible searching API we had some time ago. Should there be:

- single SQL-like parameter (with proper escaping!),
  - e.g. query=age>10 and (name=John or company=Foo)
- multiple parameters for each criteria,
  - e.g., age=10&name=John&company=Foo (but how to implement OR operator?)
- most bizarre: stateful POST on /searches with criteria modeled with JSON-like structure, returning URL to search results (/searches/25), which can later be queried

REST is really constraining here.

CRUD by definition

RESTful web services are CRUD-oriented, rather than business- or transaction-oriented. Countless times we had to carefully map business terms into simple *create*/*update*/*delete* actions. World isn't that simple and not everything can be simply described in *create or update* sentences. And even if it can, often RESTful endpoints are awfully awkward and artificial. Do you still remember POST to /searches? Moreover not all data can be mapped into URI tree structure and often we allow denormalization, e.g. the same resource is available under multiple URIs so that it's easily accessible.

Imagine a system that publishes domain events, arbitrary state changes, e.g. LoanApproved, EmailSent, etc. Of course every event has its own, distinct set of attributes. How do you design a RESTful API that consumes such events? Inevitably you'll end up with POST /domainEvents that accepts arbitrary JSON, maybe with some tag like "type": "LoanApproved". So you have an endpoint that takes arbitrary JSON "BLOB" and most likely has a giant switch-like statement to interpret various types of events properly. Replace "type" with "method" and you just reinvented JSON RPC. How would you do that in an idiomatic way? Translate every domain event on the publisher side to appropriate API call? Doesn't sound very robust.

HTTP verbs aren't descriptive enough

Mapping from business terms into POST/PUT/PATCH/DELETE/HEAD is tedious and childish. Just like with URIs, world is much richer and not everything fits into these buckets. These verbs were designed to interact with documents and forms in World Wide Web. Using REST feels like degrading our business domain back into database browser. There is no logic, no domain-driven design, no rich flows. We simply manipulate objects, shift them back and forth. Of course REST don't have to and shouldn't map directly to database entities. But even if they map to your core business domain, you still have to view your domain through limited CRUD interface.

Mixing HTTP status codes with business replies

Idiomatically signaling business errors via REST should use 4xx class of status codes. However these errors were not designed to signal business cases so I constantly find myself trying to map 4xx codes to business results. CAPTCHA test failed? Let's make it 400 *Bad request*. Form validation error? 417 *Expectation failed*? Constraint violation due to duplicate? 409 *Conflict*? But what about optimistic lock exception? What if client has insufficient balance? What if... These error codes are designed for document retrieval, not rich business behind back-end. You end up with putting everything under the same status code or building complex translation document. We invented exceptions, faults, Either<T> - just to suddenly limit ourselves to fixed set of numeric error codes. 404 is especially problematic, because it's so prevalent. When using RESTful API you can never tell whether 404 is a business circumstance or just a typo in URL. Sounds like a recipe for distributed debugging hell. Oh, and did I mention there is no standard way of encoding errors?

Temporal coupling

RESTful APIs are really popular among microservice fanboys. But let's go back to basics. RESTful APIs are based on HTTP, which is a request-response protocol built on top of TCP/IP. TCP/IP builds a connection abstraction on top of packet-oriented IP protocol. IP is barely capable of delivering messages from one machine to another. By building all these levels of abstractions we forgot that web is really asynchronous. We believe that by using contract-less lax JSON we are no longer coupling systems together. But there is another dimension of coupling: temporal dependency. If one system needs to notify another about some event, do they really need to exist at the same time. In some cases, typically implemented with GET, request-response makes sense. But in most cases what we really want is a fire-and-forget, at-least-once semantics. Message-driven distributed architectures proved to be much more robust and fault-tolerant. Two systems no longer need to see each other and live at the same time. Moreover one system will no longer break another one if it's producing too many requests. Simply put a persistent queue in between two systems. This approach has numerous advantages:

- producers and consumers can be restarted at any time without data loss or quality of service degraded
- scalability is easier to achieve, no need for complex load balancing
- we can dispatch messages to multiple systems at once
- debugging *might* be easier with wire tap pattern

RESTful also has no notion of one-way requests. POST with no body is as close as you can get. This is problematic, because many business cases are naturally fitting the fire-and-forget semantics. When I'm publishing domain events I don't care about response,

but REST services are too tightly coupled to HTTP protocol. Only typical request-response interactions (like retrieving a user by ID) aren't a good fit for queues though. Temporal queues with correlation IDs are awkward and introduce a lot of latency.

No standards

There are no standards but only good, sometimes contradicting practices. We can't even agree whether resources should be singular or plural, not to mention paging parameters, error handling, HATEOAS... You will find people arguing how RESTful is your service, according to Richardson Maturity Model. Lack of standards mean that everyone can name their service RESTful. It might be a live example from Roy Fielding's dissertation, it might be a <form> POST handler - as long as they use HTTP, they are REST. This also means you never now how to properly interact with any API. Which headers you should use, how to decode the response, how content type is negotiated (headers? extension in URL?) and which types are supported.

Backward compatibility

RESTful services have few flawed ways of handling backward compatibility:

- only adding fields, never removing or changing. I witnessed a situation where someone fixed an innocent typo in an object that happened to be directly encoded as JSON. This crashed another system
- Content type with versioning - painful and requires maintaining multiple versions
- HTTP redirects if resources are renamed - only works if clients are RESTful enough, avoiding hard-coded URLs entirely

Each technique above has its own issues, RESTful services simply aren't designed to be evolving.

Alternatives

Few questions I want you to ask yourself before jumping onto JSON over HTTP a.k.a. REST hippie-style of integration:

- is performance a concern? Then choose more compact format that doesn't require costly compression
- do you really need blocking request-response? If not, consider message queue or store, like Kafka
- are you doing continuous deployment or auto-scaling? Then pick technology that doesn't couple client and server temporarily and is connection-less, see above
- are your interactions complex or maybe you are extracting existing API/interface from module to distributed service? Then choose technology closer to classic RPC
- do you need exactly-once semantics? If so, nothing will help you, sorry