Développement par l'API

MIX-IT 2013

Éric Daspet - David Larlet

La présentation est disponible en ligne, vous aurez tous les liens.

L'adresse sera proposée en fin de conférence

Qui sommes-nous ?

Éric Daspet

- TEA, livre num.
- PHP 5 avancé
- Performances web
- Paris-Web

eric.daspet.name

David Larlet

- scopyleft
- artisan
- geek
- citoyen

larlet.fr/david

nous n'enseignons pas,
nous échangeons nos expériences

Mais nous n'avons que 50 minutes, si nous voulons vous et nous tirer quelque chose de cet échange, il faut participer

Nous présentons une question ou problématique, écoutons vos expériences, présentons nos réflexions, échangerons à la fin pour la vue globale

Qu'abordons-nous

- Par quoi commencer ?
- Comment structurer ?
- Quels formats ?
- Comment ajouter une nouvelle version ?
- Que faire des erreurs ?
- Comment sécuriser ?
- Comment gérer les spams et les abus ?
- Comment paginer ?
- Hypermédia et découverte, késaco ?
- Quelles autres bonnes pratiques ?

Nous avons préparé pas mal de sujets à aborder, mais en 50 minutes nous ne pensons pas en faire plus de 6

4 sujets ont été soumis au vote par avance
nous en gardons 2 à décider ici en fonction de nos échanges

mais nous avons des sujets en réserve si nous avons le temps

Par quoi commencer ?

Mobile first ?

Pourquoi penser mobile first ?
* parce que le desktop n'est qu'une extension du mobile ?
* parce que les contraintes techniques (performance, css, rwd) rendent cela nécessaire ?
* parce que c'est la cible la plus importante en % ou en valeur ?
* parce qu'on est plus proche des données principales à afficher ?

Si vous avez lu le sujet, vous savez qu'on vous proposera de l'API-first. Pourquoi ?

API First

D'abord mener la réflexion sur les données
et sur comment elles seront utilisées

Penser l'API en prenant la place de l'utilisateur

Et si vous deveniez votre propre utilisateur ?

Éventuellement en commençant par le mobile
ou y brancher des applicatifs indépendants

1. réfléchir aux données et à leur usage
2. penser l'interface (programmative)
3. développer l'accès au données
4. publier cet accès aux données
5. proposer une présentation simple du contenu
6. enrichir pour la vue mobile
7. enrichir pour la vue web
8. recommencer au 1 pour une évolution

Comment structurer ?

Appel, URL, ressources, hiérarchies

Structure

REST, fuyez SOAP, évitez le "truc perso"
Ne vous limitez pas aux verbes, utilisez HTTP

Une ressource = un nom, en minuscules

hiérarchies à plat, /ressource/ et /ressource/id
profondeur limitée à /ressource/id/ressource

vs. approche liée (hypermedia)

Utiliser le web
Pensez à vos usages, pas forcément comme un site web

REST :

Un verbe pour l'action, c'est la méthode HTTP. Ce peut être GET, POST, mais pas seulement.
Il y a bien plus, et ce n'est pas limitatif (même si probablement suffisant)

Une racine (protocole, nom de la machine, adresse racine du service)

L'adresse relative de la ressource
* en minuscule (simplifiez la vie de vos utilisateurs)
* une ressource = un nom (et pas un verbe), sinon vous faites erreur quelque part
* prévoyez des noms concrets, évitez le sur-générique qui n'a plus de sens pour personne (quelqu'un connait WCAG2 ?)

Les hiérarchies à rallonge sont vite agaçantes lors des tests et développement des clients
mais surtout elles vont limitez l'extensibilité
généralement "catégorie de ressource" + "identifiant de la ressource dans sa catégorie" suffisent
parfois on rajoute un sous-type de ressource : /livre/12345/sommaire

Quel(s) format(s)

JSON, ATOM, XML, (x)HTML, RDF, www-form-urlencoded, ...

Format

Requêtes : pensez au form-encoded

Utilisez des formats standard !
Utilisez un format extensible pour plus tard

Utilisez une extension de fichier dans l'URL
Déclarez le format dans les entêtes
Imposez le format, et le codage (UTF8 svp)

Le mot d'ordre tout le long : Faites simple !

Pour les requêtes, passer les paramètres dans l'URL (GET) ou le corps (POST) est tellement standard et facile à utiliser avec les outils basiques (formulaires, CURL, ...) qu'il vous faut une bonne raison pour faire autre chose

Pour les réponses, deux choses importantes :
* un format standard et simple
* un format qui permette d'ajouter des données des extensions sans impacter les clients

Impacts :
* Côté client : ne pas avoir une liste restrictive des clefs ou balises présentes, accepter et ignorer ce qu'on ne connait pas
* Validation type xmlschema : la fausse bonne idée qui tue toute évolution
* Côté serveur : exemple json : n'envoyez pas directement un tableau de résultat, mais un objet qui contient vos résultats, pour pouvoir ajouter d'autres éléments de contexte plus tard

Pensez aux formats pré-existants. Pour des listes, ATOM est excellent car déjà supporté par une batterie d'outils. Pour des documents utilisateurs, HTML avec du rdfa ou microdata est un peu plus difficile à analyser mais peut être affiché tel quel et permet d'utiliser un navigateur + des formulaires pour naviguer dans l'API.

Comment faire une nouvelle version ?

v1.2.4-beta

Versionnement

Restez compatible : même URI, ajout de paramètres optionnels, valeurs additionnelles

Si besoin d'une nouvelle version :

- uniquement des versions majeures (v1, v2)
- /v2/* en racine du service
- rarement dans le chemin de la ressource

vs. approche liée (hypermedia)

La règle est la compatibilité, parce que vous ne pouvez pas gérer les clients

Si vous avez bien choisi vos formats et structures de données, vous pouvez ajouter de nouveaux paramètres optionnels dans vos requêtes, et de nouvelles données dans vos réponses, le tout sans avoir à toucher à la compatibilité avec vos anciens clients.
Au fur et à mesure vous pourrez déclarer obsolète quelques usages, mais votre API évoluera en continue sans rupture forte.

Si vraiment besoin (pas plus d'une fois tous les 1 ou 2 ans), vous pouvez faire un big bang avec une nouvelle version majeure. Gardez cependant un maximum de point d'entrée rétrocompatibles

Faites simple : un /v2/ en racine du service sera simple à comprendre par les développeurs, et simple à gérer côté serveur.
Ne faites que des versions majeures. v2, v3, v4. Le reste est inutile vu que les mineures sont sensées être rétrocompatibles assez longtemps.

Si votre nouvelle version se limite changer le fonctionnement d'une ressource, vous pouvez éventuellement faire un v2 sur l'url de la ressource elle-même. Mais attention

Que faire des erreurs ?

Erreurs

Utilisez les erreurs HTTP
au minimum 2xx, 3xx, 4xx, 5xx

Détail dans le corps du message respectant le format attendu

Code + lien vers l'aide en ligne correspondante
message : pour dev ou pour utilisateur final ?
différenciez bien les deux

Toujours renvoyer les erreurs sous forme HTTP

Rails a une bonne liste de codes d'erreurs, les officiels et les autres relativement standard
http://www.codyfauser.com/2008/7/4/rails-http-status-code-to-symbol-mapping

Détail dans le corps de la réponse, en json si vous attendez du json, en xml si vous attendez du xml, etc.

Toujours mettre un code, pas qu'un texte d'erreur
S'il y a message, déterminer s'il est pour le développeur ou pour l'utilisateur final

Attention à la gestion de l'internationalisation si message pour l'utilisateur final

Comment sécuriser ?

Sécurité

Utilisez ce que vous maîtrisez

Toujours du SSL, toujours valider les certificats

Auth HTTP basic fonctionne très bien
Auth déléguée : OAuth 1, attention à OAuth 2
Évitez les cookies de session (CRSF)

Demandez une clef d'api par applicatif

La meilleure solution sera celle avec laquelle vous êtes à l'aise

Réinventer la roue c'est se tirer une balle dans le pied. Ça fonctionne à tous les coups, même de très gros acteurs avec plus du milliers d'ingénieurs plus compétents que vous s'y sont laissés avoir.
Si vous avez à toucher vous même des primitives de chiffrage ou hachage ou si vous définissez des protocoles, c'est mauvais signe.

La base : toujours passer par du SSL. Sans ça vous mettez à risque vos clients.
Corollaire : toujours vérifier la validité des certificats, y compris dans vos codes exemples. C'est le problème le plus courant côté client.
Oui, les autorités de confiance ne sont pas parfaites, mais ce n'est pas si mal si vous n'êtes pas une cible internationale, et même là appréciable

Faire simple : HTTP basic c'est tout à fait fiable, efficace. Il faut juste penser à faire un stockage intelligent des mots de passe côté serveur (utilisez une bibliothèque pour ça, si vous faites vous même du MD5 ou du SHA1 c'est mauvais signe)

Si vous avez un système à trois parties (le client, le serveur et l'utilisateur), alors OAuth est une bonne piste. OAuth 1 est fiable, OAuth 2 est plein de pièges qui nécessitent une expertise que vous n'avez probablement pas, même des société comme Facebook s'y perdent, même l'auteur de la spec la déconseille.

Comment gérer les spams et abus ?

SPAM et abus

Throttling/métriques

- par IP
- par application (clef d'API)
- par utilisateur

Coûteux en ressources, mais indispensable

Quid d'imposer de toujours être authentifié ?

Limiter la taille des résultats

Timeout contre les requêtes longues

Si vous ne prévoyez pas dès le départ de quoi limiter le nombre de requêtes, cela peut être cher après coup

Cela coute forcément des ressources de compter des requêtes, mais ça coutera encore plus cher de ne pas le faire

* Compte par IP, pour limiter le spam

* Compte par application, pour éviter les applications infectées, ou simplement mal programmées qui saturent en requêtes

* Compte par utilisateur, pour éviter les comportements abusifs

Imposer l'authentification permet de mieux définir les contraintes
Imposer au moins la clef d'API, même si c'est peu fiable

Taille des résultats : ne pas laisser les gens requêter un historique de 50K items d'un coup

Les timeout sont indispensables, autant pour éviter de faire partir votre serveur dans un traitement trop long, que pour vous protéger contre les clients mal programmés ou les DoS.

Comment paginer ?

Pagination

Faire simple ? pas ici, pièges en vue

Cas simple : sort + order + limit + offset
Hypermedia : liens rel=next, rel=prev

Quid des nouveaux items entre 2 requêtes ?
exemple avec twitter :
paramètre since, upto avec une date ou un id

Cas simple :
* limit : nombre de résultats
* offset : nombre d'items à ignorer

Impose un sort et order pour un critère de tri (éventuellement obligatoire) et un ordre de tri
Penser à plafonner le nombre de résultats par page et la profondeur de l'offset (ex: google, twitter)

Non efficace pour les résultats qui peuvent changer entre deux requêtes
Bonne pratique : utiliser since ou upto à partir d'une date ou de l'identifiant d'un item, pour que la requête fonctionne de façon cohérente même si la liste d'item évolue

Attention à toujours plafonner la pagination,
Éventuellement ne pas autoriser tous les types de tris : faites des choix !

Hypermedia: rel=next, rel=prev

Hypermédia et découvertes, késaco ?

Hypermédia et découverte

Envisager votre API comme votre site Web : navigable, standard, indexable, évolutif.

Ne plus fournir des ids mais des liens

Utiliser un format qui permette d'utiliser l'hypertext (pas JSON par exemple)

Un point d'entrée unique devrait suffire

Ne pas s'attendre toutefois à une trop grande utilisation du système

La plupart des développeurs recopieront un code qui fonctionne, ou feront la découverte une fois manuellement et coderont tout en dur

Le système de liens pour référencer les sous-ressources est toutefois une bonne idée simple à appliquer. Exemple avec ATOM

Permet de naviguer sur toute l'API depuis la page d'accueil

Quelles bonnes pratiques ?

Bonnes pratiques

Soyez en relation avec vos utilisateurs

Dates au format ISO, *avec* fuseaux horaires

Utilisez des codes, pas des messages texte

Hypermedia : documentez les types de liens

Améliorer les performances a posteriori
(préchargement des sous-ressources)

Chaque option ajoute de la difficulté de maintenance et d'évolution. Faites au plus simple;

Le fuseau horaire est l'écueil le plus classique. Prévoyez toujours des timezone

Pensez à l'internationalisation, et à l'automatisation : Utilisez des codes, pas des messages texte

Certains services ont un paramètre "fields" pour lister les champs intéressants à retourner dans la réponse
permet de lister les sous-objets à inclure
mais attention, on complexifie le serveur
fields=description,related(title,date),...

Documentez les concepts et les objets métier qui sont manipulables, pas les URL ni les verbes HTTP. Votre site web nécessite-t-il une documentation avant de l'appréhender ?

Améliorez les performances a posteriori, découvrez les chemins les plus souvent empruntés et fournissez des raccourcis, pré-chargez les sous-ressources couramment consultées.

Pas de questions

Normalement c'est déjà fait :-)

Quelques retours sur le format utilisé ?