MEETUP PLOSS - 15 FÉVRIER 2016

LE FRONT-END EN 2016

Objectifs

- Discussion libre sur les évolutions du développement *front-end* (principalement web)
- Quelques observations et questions ouvertes pour lancer le débat
- Focus sur les opportunités et les risques économiques

Quelques tendances (insistantes ou Emergentes)

- L'explosion cambrienne des *frameworks* JavaScript
  - MVC vs. *reactive programming*
  - L'approche par composants
    - Quel avenir pour les frameworks d'UI riche ?
- Maturation de l'outillage (langages et outils)
- Technos web et développement mobile
- Architecture: REST vs *Demand-Driven*
- Autres ?

Explosion des frameworks JS

- **Promesse économique**: innovation rapide, richesse du choix
- **Risque économique**: obsolescence tout aussi rapide
- Historique (partiel)
  - jQuery
  - Backbone (+ éventuellement extensions)
  - Ember
  - Angular 1 puis 2
  - React
  - Alternatives récentes: vue.js, riot.js, aurelia, mithril…
- Nombreuses blagues d'informaticiens sur les frameworks Javascript

Ryan Bates
@rbates

Follow

I'm starting to wonder if there are more client-side JavaScript frameworks than there are apps that use them.

Reply Retweet Favorite More

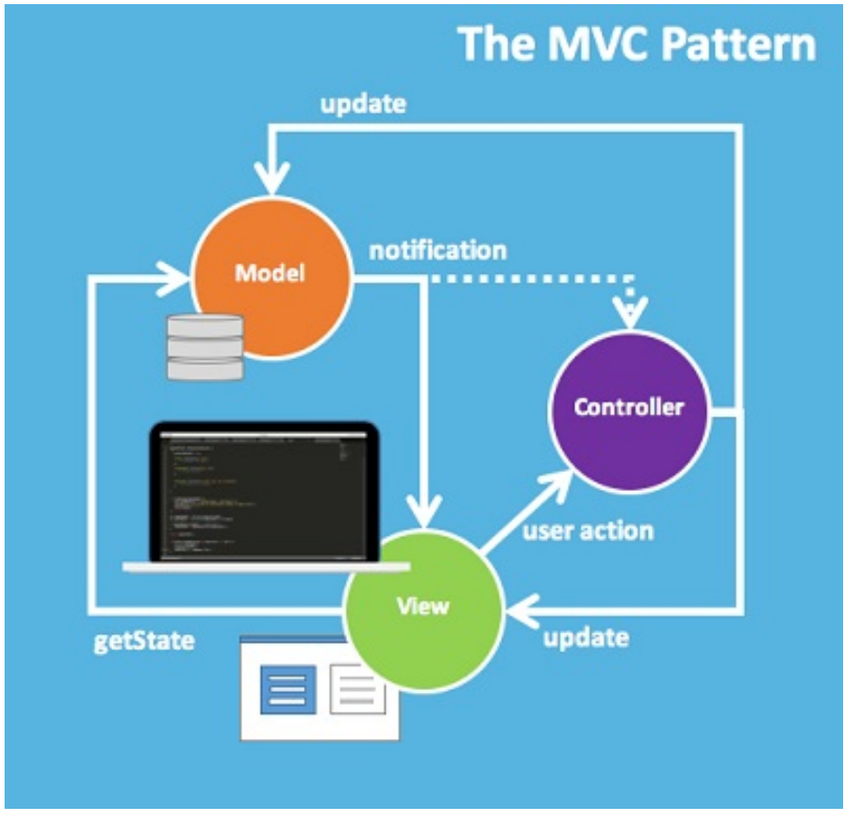
RETWEETS	FAVORITES
330	52

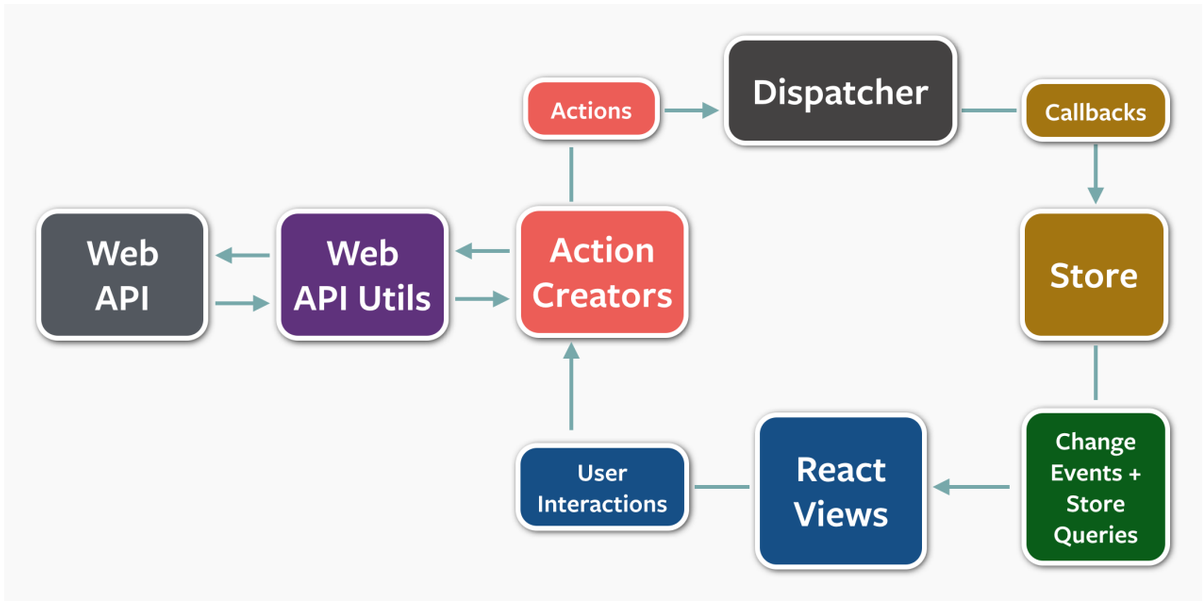
8:38 AM - 25 Sep 2012

MVC vs Reactive

- MVC
  - *Pattern* bien connu depuis les années 80 (+ variantes plus modernes: MVVM, MTP, etc.)
  - Ex: Backbone+extensions, Angular, Ember…
    - *2-way databinding* avec Angular 1, Vue et quelques autres
- "Reactive": focalisés uniquement sur la couche "view"
  - Mode lancée par React (Facebook), suivie par riot.js, vue.js, preact, etc.
  - Architecture Flux + variantes, Elm, RxJS+extensions…
  - Quelques avantages indéniables (slide suivant)

MVC vs. REACT+FLUX





Les promesses de React & friends

- Plus simple de raisonner sur ce qui se passe dans l'application
  - Notamment si on passe par la notion d'immutabilité
- Architecture découplée
- Hot-reload avec conservation de l'état de l'application
  - Avec l'outillage adéquat
- Surface d'API plus restreinte (=> charge cognitive plus faible sur les développeurs, une fois qu'on a compris le modèle!)

Approche par composants

- **Promesse économique**: favoriser la réusabilité du code, en interne ou dans le cadre de bibliothèques publiques
- Tous les frameworks modernes se réclament de cette approche
  - Succès indéniable des plugins jQuery
  - Angular, React, Vue, Aurelia…
- **Risque**: non interopérabilité des approches actuelles
  - 1 standard émergent: Web Components (+ une impl, Polymer)

Quel avenir pour les frameworks complets ?

- Les *frameworks* récents (Angular, React & friends) ne proposent pas de bibliothèques riches de composants UI prêts à l'emploi
- Contrairement à la génération précédente:
  - YUI, Ext, Kendo UI, SmartUI, Closure Library, etc.
- Certes, il est plus facile de développer des composants dans ces nouveaux *frameworks*
- Mais, à ce stade, moins facile de se servir sur étagères (IMHO)

Maturation de l'outillage

- **Promesse économique**: industrialiser le dev *front-end* pour plus de qualité
- Outils de tests (mocha, jasmine, karma, etc.), d'analyse statique (jslint, eslint)
- Outils d'automatisation de la chaîne de production (grunt, gulp, npm+webpack)
- Outils de gestion des dépendance (ex: npm, bower en train de passer de mode)
- Outillage des navigateurs (déboggeurs, traceurs, explorateurs de composants)

Outillage: transpilation vers JavaScript

- **Promesse**: s'affranchir au plus vite des limitations de JavaScript ("the bad parts")
- JS->JS: traceur, Babel (ex: 5to6)
- Langagues alternatifs: Clojure, Elm, Typescript, ClojureScript, js_of_ocaml, même Python ;)
- Focus souvent sur les paradigmes fonctionnels
- Chaque langage a des promesses différentes (typage fort, immutabilité)
- Intégration dans les chaînes de production, support (relativement récent) des navigateurs (debuggers)

Technos Web et développement mobile

- **Promesse économique**: mutualisation du code entre les plateformes (iOS / Android / autres)
- Historique
  - Les ancêtres: jQuery Mobile, Sencha Touch, Appcelerator Titanium
  - Ionic (basé sur Angular)
  - React-native
  - Autres ?
- Tendance aussi récente sur le desktop avec Electron (fork de node-webkit)

Architectures REST et demand-driven

- **Promesses économiques:**
  - S'imposer des contraintes architecturales pour découpler proprement les développement *front* et *back*
  - S'appuyer sur des bibliothèques ou des *frameworks* déjà établis sur le *back-end* (et plus émergents sur le *front-end*)
    - Serveur: Flask-Restful ou DRF (Python), JAX-RS (Java)…
    - Client: Ember Data, Breeze…
  - Tenir compte des évolutions architecturale du Web

REST ET LE MODÈLE DE MATURITÉ DE RICHARDSON

Richardson Maturity Model (RMM)

- **Level 3**
  - Level 2 + Hypermedia
  - RESTful Services
- **Level 2**
  - Many URIs, many verbs
  - CRUD services (e.g. Amazon S3)
- **Level 1**
  - URI Tunneling
  - Many URIs, Single verb
- **Level 0**
  - SOAP, XML(POX), RPC
  - Single URI

REST: CHALLENGES

- “Standardisation”: nombreux protocoles concurrents, principalement autour de JSON
  - HAL (http://tools.ietf.org/html/draft-kelly-json-hal-07)
  - JSON API (http://jsonapi.org/)
  - Restful Objects (http://www.restfulobjects.org/)
  - "collection.document+json” (http://cdoc.io/)
  - "collection+json” (http://amundsen.com/media-types/collection/)
  - OData (www.odata.org), soutenu par Microsoft et SAP…

REST: CHALLENGES (2)

- HATEOAS (*Hypermedia as the Engine of Application State*, i.e. le niveau 3 du RMM) n'est pas simple à mettre en oeuvre côté client (un framework dédié peut aider)
- Et surtout: le serveur doit tenir compte de toutes les demandes possibles de la part des clients
  - Performances: requêtes multiples pour réaliser une seule page Web
  - Challenges pour les équipes de développement
    - L'équipe *front-end* doit demander les changements
    - N équipes *front end* qui attaquent un seul *back end*

Architectures Demand-Driven

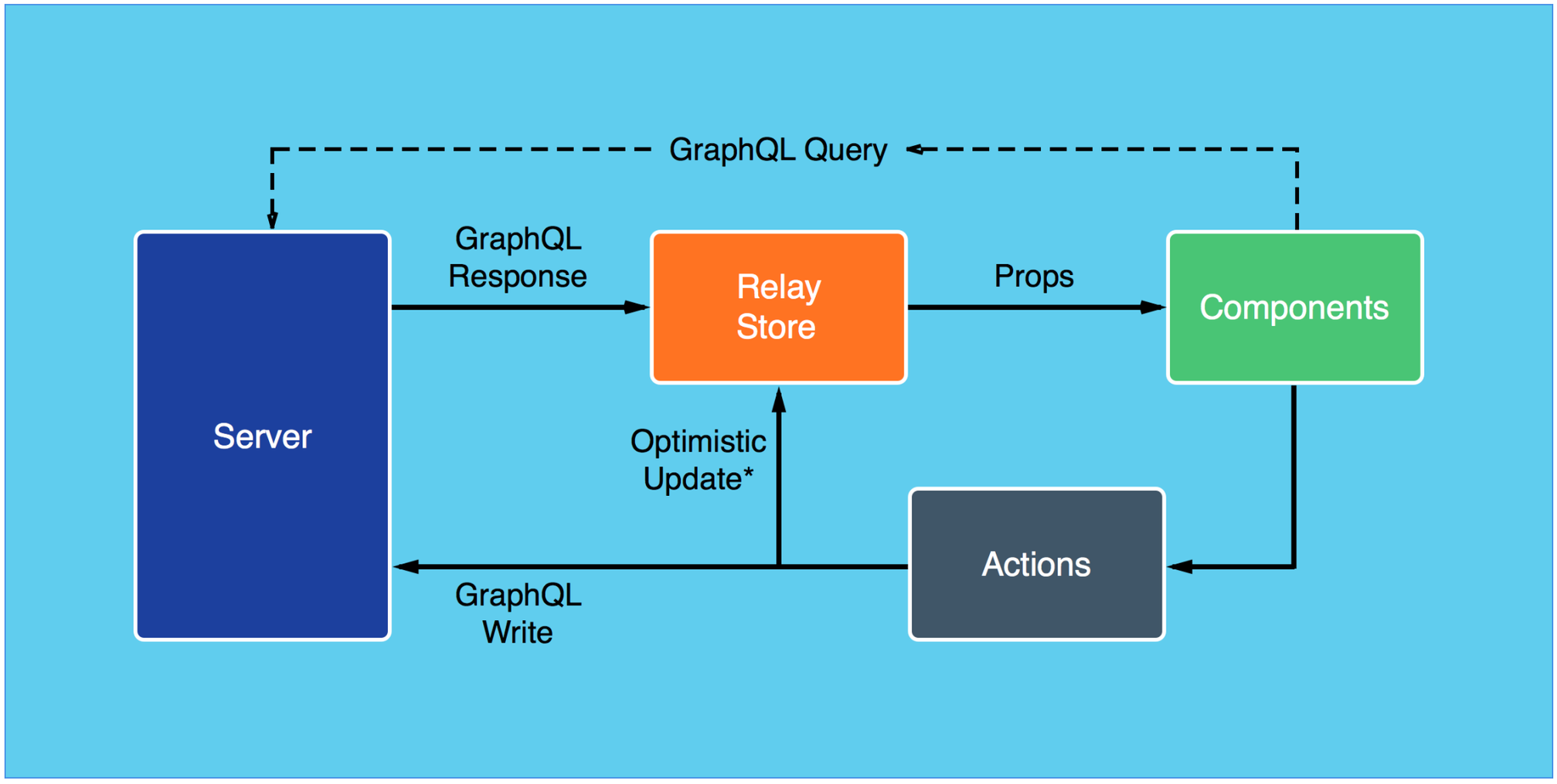
- Paradigme émergent (1 an 1/2) poussé par Netflix (Falcor) et Facebook (GraphQL + Relay), ou la communauté Clojure (Om)
- **Promesses**:
  - Découplage complet du *dev front* vs *back* grâce à un langage et/ou un protocole de requêtes dédié
  - Transmission de données hétérogènes dans une seule requête
  - Mises à jour optimistes
  - Synchronisation intelligente, mode déconnecté

Exemple (GraphQL)

```
{
  user(id: 4802170) {
    id
    name
    isViewerFriend
    profilePicture(size: 50)  {
      uri
      width
      height
    }
    friendConnection(first: 5) {
      totalCount
      friends {
        id
        name
      }
    }
  }
}
```

```
{
  "data": {
    "user": {
      "id": "4802170",
      "name": "Lee Byron",
      "isViewerFriend": true,
      "profilePicture": {
        "uri": "cdn://pic/4802170/50",
        "width": 50,
        "height": 50
      },
      "friendConnection": {
        "totalCount": 13,
        "friends": [
          {
            "id": "305249",
            "name": "Stephen Schwink"
          },
          {
            "id": "3108935",
            "name": "Nathaniel Roman"
          }
```

Exemple d'architecture (GraphQL)



GraphQL: déjà des solutions

- JavaScript: GraphQL (Facebook)
- Python: Graphene
- Java, PHP, etc.

Discussion

- L'explosion cambrienne des *frameworks* JavaScript
  - MVC vs. *reactive programming*
  - L'approche par composants
    - Quel avenir pour les frameworks d'UI riche ?
- Maturation de l'outillage (langages et outils)
- Technos web et développement mobile
- Architecture: REST vs *Demand-Driven*
- Autres sujets ?