Le realtime avec React

Le realtime dans le web

A toujours été en général très compliqué...

AJAX

Tout a démarré de là, en 1998, alors que XMLHTTPRequest était un composant ActiveX de Microsoft.

AJAX permet de faire des requêtes asynchrones pour récupérer des données sans recharger une page.

Problème:

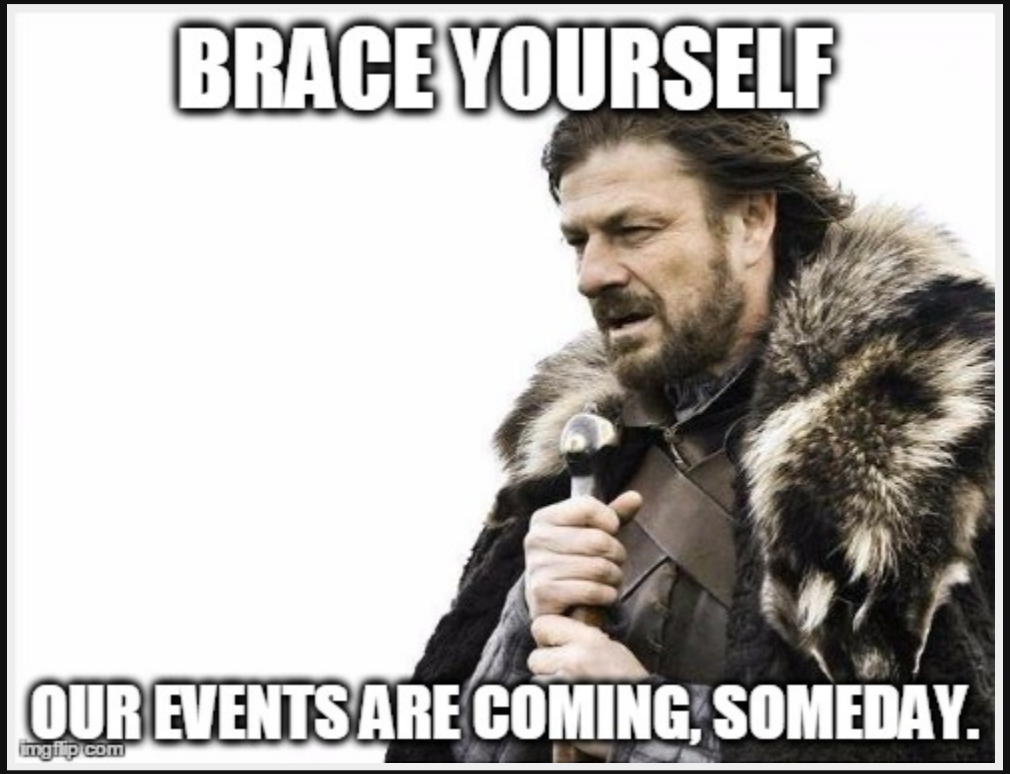
- Il faut que le client initie la requête.

Polling

Le polling consiste à ouvrir une connexion et attendre que le serveur y répond.

Problèmes:

- Impossible à faire scaler sans difficulté
- Latence très haute

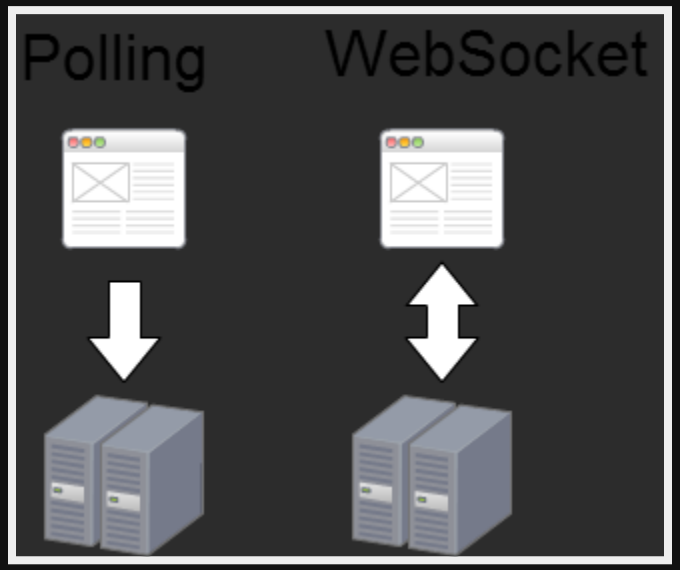

BRACE YOURSELF
OUR EVENTS ARE COMING, SOMEDAY.
imgflip.com

La naissance des WebSockets

Les WebSocket sont un nouveau standard normalisé assez récemment.
Cette technologie est né afin de pallier les **hacks** qu'on utilisait, le Long Polling notamment.

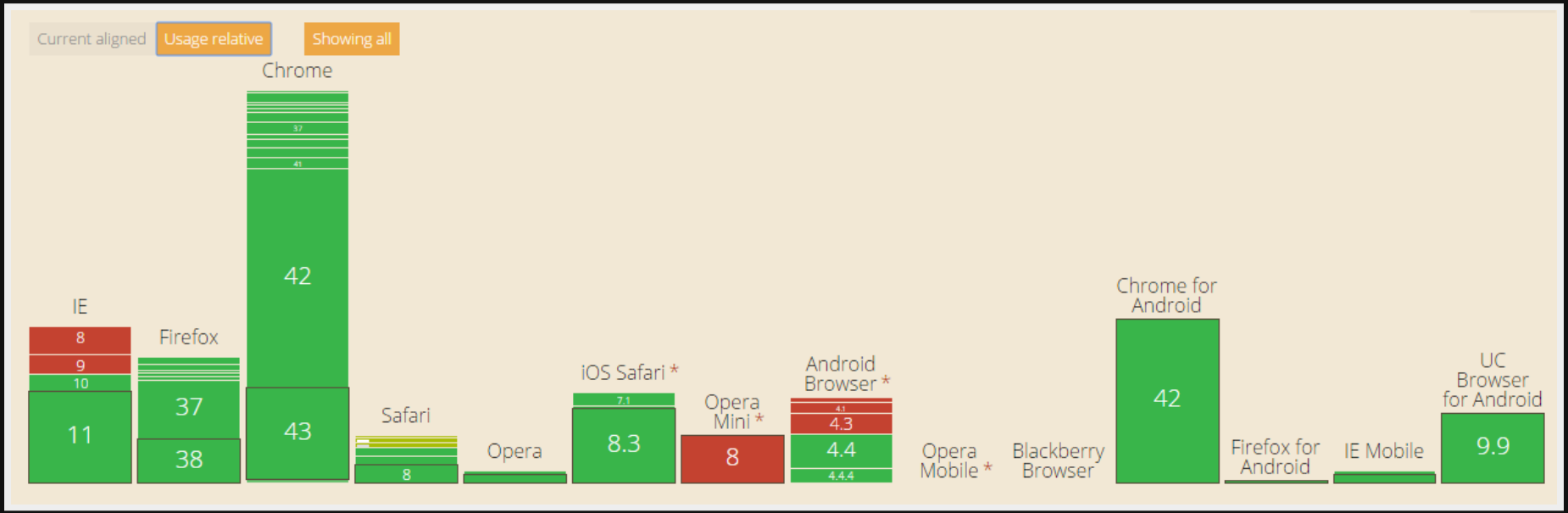
C'est un canal de communication bidirectionnelle, sur une socket TCP.

Pourquoi les WebSockets par rapport à AJAX?



Qu'en est-il du support?

Il se trouve que c'est pas mal du tout, seul Opera Mini 8 ne supporte pas les WebSockets.



Socket.IO, le level suivant

Socket.IO encapsule les WebSocket et ajoute une abstraction qui fait en sorte que Socket.IO fallback sur du polling ou des Flash Sockets en absence de WebSockets

En plus de cela, il offre du PubSub intégré (avec d'autres trucs cool).

```
var io = require('socket.io')(80);
var cfg = require('./config.json');
var tw = require('node-tweet-stream')(cfg);
tw.track('socket.io');
tw.track('javascript');
tw.on('tweet', function(tweet){
  io.emit('tweet', tweet);
});
```

Mais... Peut-être qu'il y'a trop de boilerplate encore ?

Socket.IO est très sympathique, beaucoup de projets l'utilisent activement.

Cependant pour une grosse web app en SPA, ça peut être trop léger?

Comment on fait pour gérer l'authentification?

Comment on fait pour gérer l'autorisation?

Beaucoup de comment (et de boilerplate) !

Meteor.js, le fruit ultime du realtime

Il s'agit là d'un des framework web les plus avancés en terme de realtime, il fonctionne en imposant quelques règles:

- Isomorphique par définition, JS server-side, JS client-side.
- Un nouveau écosystème de packages.

En échange de ça, il vous donne la possibilité de:

- Live updates (realtime)
- Compensation de la latence (Interface optimiste)
- Déploiement à chaud

En plus, ça s'intégre bien avec React !

```
var MyComponent = React.createClass({
  mixins: [ReactMeteor.Mixin],

  startMeteorSubscriptions: function() {
    Meteor.subscribe("players");
  },

  // Make sure your component implements this method.
  getMeteorState: function() {
    return {
      playerCount: Players.find().count(),
      ...
    };
  }
});
```

Une alternative à Meteor.js

WAMP, une approche différente

WAMP, une approche différente

WAMP est un protocole qui peut être implementé sur des WebSockets, des RawSockets, ou n'importe quel type de transport.

RPC & PubSub

WAMP offre la possibilité d'utiliser des RPC et du PubSub à travers des libraries (qu'on appelle AutobahnXXX souvent).

- Du JavaScript (Node.js et Browser - IE10 et +)
- Python (2.7 avec Twisted et 3.4 avec asyncio)
- C++ (si si, faire un backend en C++ est facile maintenant)
- Java
- Objective-C
- PHP (malheureusement)
- C#
- Et plein d'autres... (des clients Rust et Elixir sont prévus !)

Des RPC

Remote Procedure Call, c'est un appel de fonction à distance.

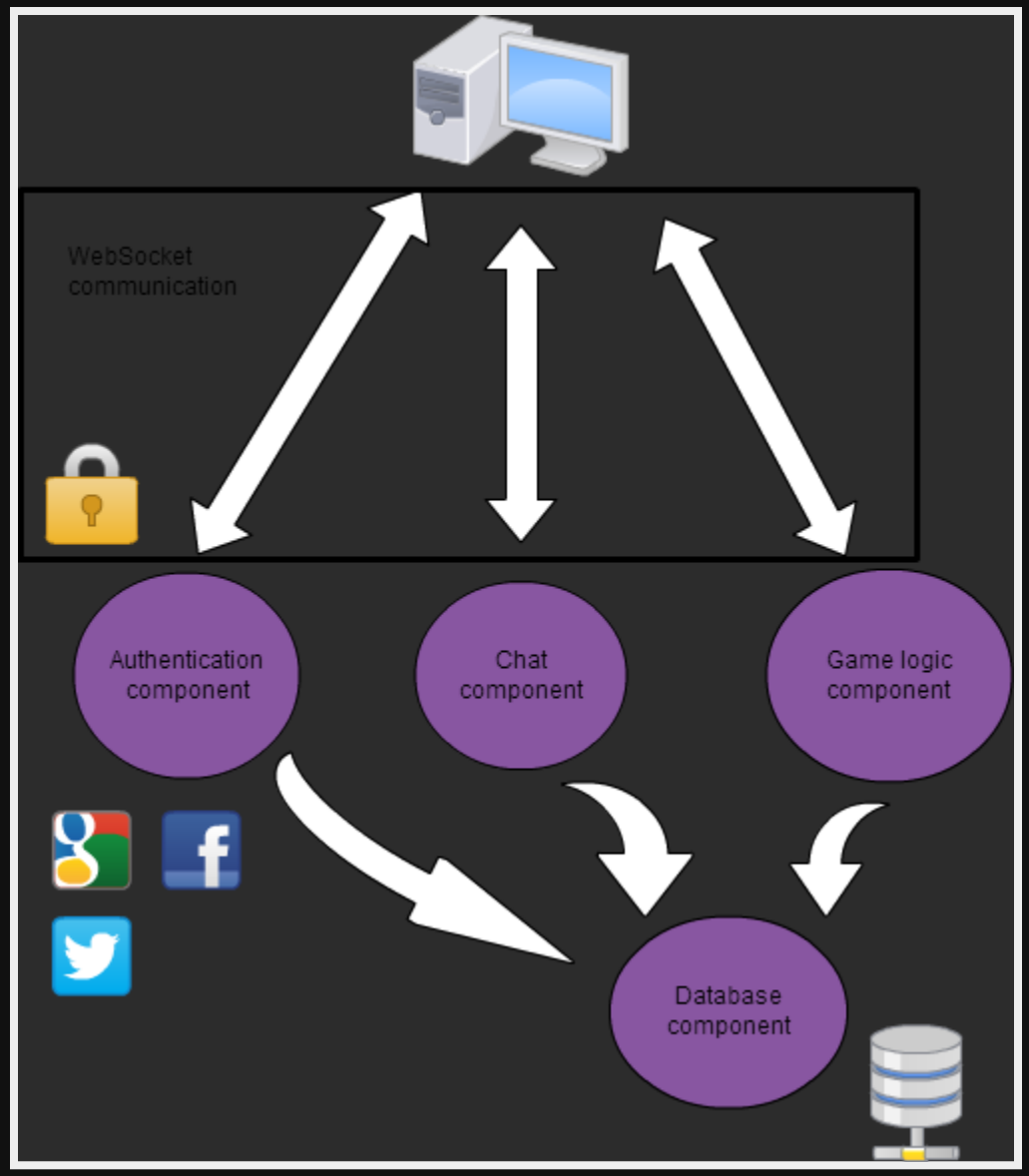
Les RPC par l'exemple

```
session.call('com.meetup.react.eat_pizza', [
    {
        "feedback": "Very good pizzas, I must say.",
        "happy": true
    }
]).then(function (pizzas) {
  console.log("Mmmmmm, good pizzas.");
}).catch(function (why) {
  console.log("Sad panda :(", why);
});
```

Principe de microservices

Chaque élement connecté au routeur peut enregistrer des RPC

Ainsi, on peut découpler les services d'authentification, de chat, de l'application, etc...



On peut facilement faire du hot swapping de services, du load balancing, et s'assurer de très bonnes performances.

Du PubSub

On peut souscrire à un topic et recevoir des évenements publiés.

On peut vice-versa publier des événements à 0, 1, 2 jusqu'à N clients qui ont souscrits à un topic.

Exemple : Subscribe

Et tout le monde ainsi peut savoir qui mange des pizzas!

```
function onPizzaEaten(args, kwargs, details) {
   if (!details.publisher) {
     console.log("I don't know who it is, but he has eaten a pizza anyway!");
   }
   console.log("Hey! " + details.publisher + " has eaten a pizza!");
}
session.subscribe('com.meetup.react.pizza_eaten', onPizzaEaten);
```

Exemple: Publish

Quelqu'un peut publier des événements à toutes les personnes qui ont souscrit au topic.

```
session.publish('com.meetup.react.pizza_eaten', [], {},
  {disclose_me: true});
```

Essayons ce morceau de code!

Exemple : Synchronisation

On peut s'en servir pour gérer des events.

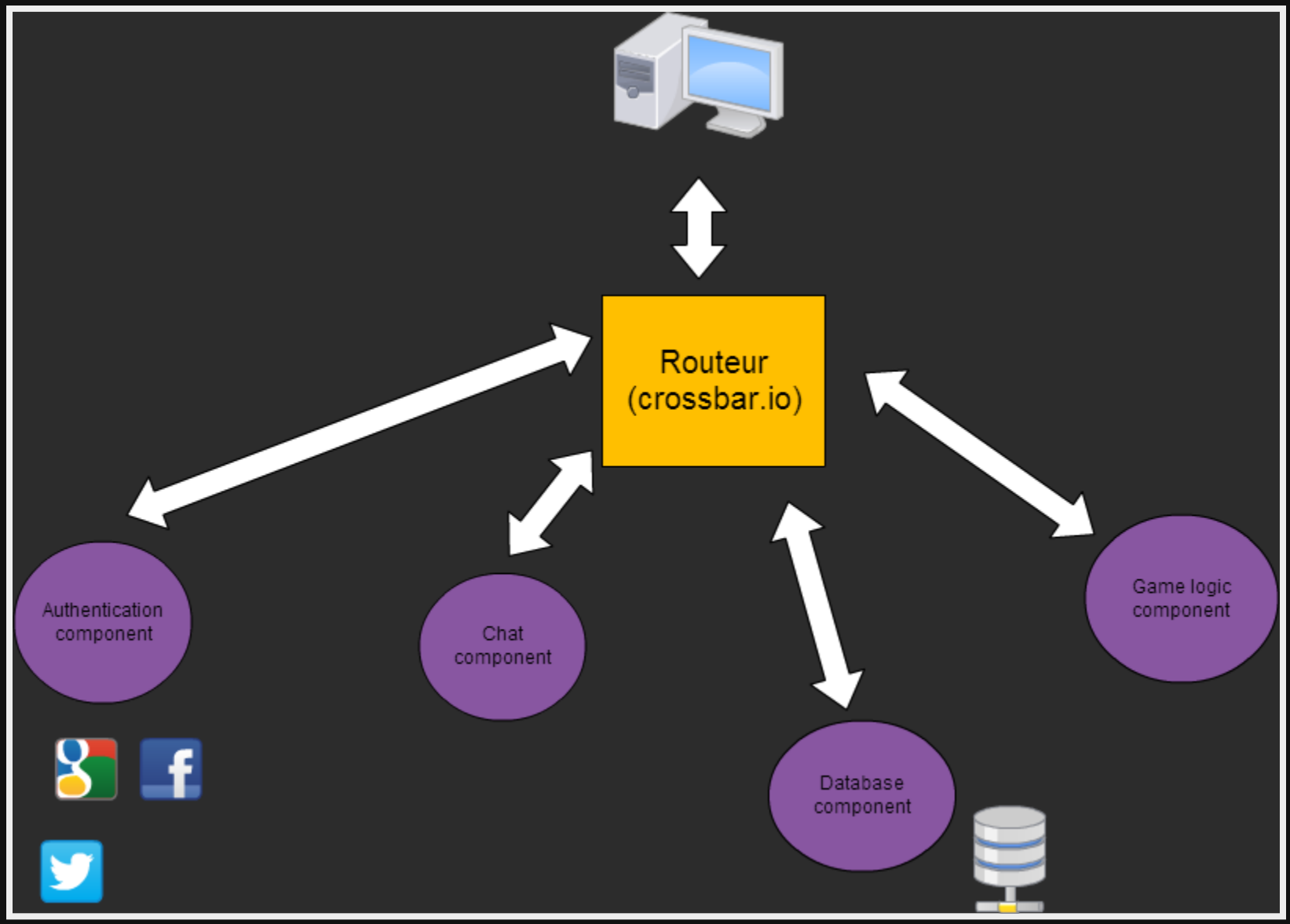
```
session.subscribe('com.meetup.react.participants.on_leave',
  function (leaver) {
    console.log("Bye bye " + leaver);
    removeParticipant(leaver);
});
```

Mais comment ça marche ?

En réalité, à chaque appel que vous faites (PubSub ou RPC), vous passez par un routeur.

Ce routeur va assurer l'interopérabilité entre les différents composants, la sécurité (authentification + autorisations).

L'implémentation du routeur existe en plusieurs langages, mais la plus avancée reste Crossbar.io (Python)


Routeur
(crossbar.io)
Authentication component
Chat component
Database component
Game logic component

Qu'en est-t-il de React dans cette histoire?

Faire du realtime avec React est possible à travers plusieurs moyens.

- Meteor.js (JavaScript only)
- Relay / GraphQL (Indisponible)
- AutobahnJS ?

AutobahnJS ?

Le problème avec AutobahnJS, c'est le code peut vite devenir complexe à cause de la boilerplate nécessaire.

```
var c = new autobahn.Connection({
  url: 'ws://127.0.0.1:8000/ws',
  realm: 'meetup',
  authmethods: ['ticket'],
  authid: 'Raito',
  onchallenge: function (session, method, extra) {
    if (method !== "ticket") {
      throw new Error("Invalid auth method!");
    }

    return "passw0rd";
}});
```

AutobahnReact

C'est une lib JS qui vise à wrapper toute la boilerplate à écrire qu'on rencontre avec Autobahn et React.

Elle est très jeune encore.

Open source en licence GPLv2.

https://github.com/RaitoBezarius/AutobahnReact

Codée en ECMAScript 6, transpilée par Babel.

Gestion de la durée de vie d'une connexion

L'initialisation

```
Autobahn.initialize('ws://127.0.0.1:8000/ws', 'example');
```

```
Autobahn.browserInitialize(8000, 'ws', 'example');
```

Cycle de vie d'une connexion

```
Autobahn.Connection.onLost((details) => {
  this.setState({connected: false,
    message: "Attention, la connexion est temporairement interrompu!"});
});
```

Authentification et autorisations

Authentification

Plusieurs schéma d'authentification sont possibles.

- WAMP-CRA
- Ticket
- OTP
- Le vôtre?

Exemple : Authentification (ticket)

```
Autobahn.Auth.become("topsecret_ticket_randomly_generated")
.then((session) => {
  console.log("You're authenticated!");
});
```

Exemple : Authentification (WAMP-CRA)

```
Autobahn.Auth.logIn({username: "raitobezarius", password: "ihatephp"})
.then((session) => {
  console.log("Welcome! You're authenticated as " + Autobahn.Auth.currentUser.role + "
});
```

Exemple : Autorisation (rôles)

```
render() {
  return (
    <RolePolicy roleMustBe="admin">
      <DeleteButton />
    </RolePolicy>
  );
}
```

Exemple : Autorisation (dynamique)

```
componentDidMount() {
  Autobahn.Auth.canAccess("com.user.124.edit")
  .then((Ok) => {
    this.setState({canEdit: Ok});
  });
}
```

Gestion des événements (subscriptions)

Malheureusement, la gestion des subscriptions reste assez tricky à gérer.

Boilerplate par l'exemple

```
componentDidMount() {
  Autobahn.subscribe('com.first_route', this.onSomethingHappen.bind(this))
  .then((subscription) => {
    this.subscriptions.push(subscription);
    return Autobahn.subscribe('com.second_route', this.onSomethingOtherHappen.bind(this
  }).then((subscription) => {
    ...
  }
}
```

Boilerplate par l'exemple (2)

```
componentWillUnmount() {
  for (var i = 0 ; i < this.subscriptions.length ; i++) {
    Autobahn.unsubscribe(this.subscriptions[i]);
  }
}
```

La solution : Le Higher Order Component

```
const ChatRoom = Autobahn.Decorators.requireSubscription(class extends React.Component
  static observeSubscriptions() {
    return {
     messages: { route: 'com.example.chat.new_message', store: true }
    }
  }

  render() {
    return (
        <div class="chatroom">
       <TextArea messages={this.props.data.messages} />
       <SendMessageBar />
       <div>
    );
  }
}, {
  willTransitionTo(transition) {
    if (Autobahn.Auth.currentUser.role !== "user") {
      transition.redirect('/login');
    }
});
```

D'autres exemples?

Vous pouvez cloner le repo et naviguer parmi le dossier examples qui contient actuellement:

- Un chat realtime
- Une application typique d'authflow

Mais c'est quoi les avantages par rapport à une REST API et du PubSub ?

Les avantages

Argument	WAMP	REST API + PubSub
La vitesse	Les WebSockets sont rapides.	Les payloads HTTP sont lourds et lents (car stateless).
La simplicité	Le RPC reste assez intuitif.	Le style REST est parfois compliqué, la sémantique porte à confusion.
L'efficacité	Le PubSub de WAMP permet des updates partielles efficaces.	Sans PubSub, ça serait des full updates.
Le choix de la sérialisation	Si le JSON est trop lourd, pourquoi pas du MessagePack ?	Tu prends du JSON ou du XML.

Les avantages (2)

Argument	WAMP	REST API + PubSub
Language agnostic	Autobahn{Cpp,JS,Python,Android,...} \o/	Certains langages sont peu utilisable pour servir des API.
Backend distribué et modulaire	Rien n'est couplé, vous pouvez déconnecter temporairement des composants par exemple.	Si un de vos systèmes plante, tout plante, il faut utiliser des load balancer,

et plein d'autres choses...

Les désavantages

Argument	WAMP	REST API + PubSub
L'immaturité d'Autobahn	WAMP V1 a été déprécié récemment, c'est encore jeune.	Le style REST existe depuis un moment.
Quelques outils doivent être améliorés	Le debugging est parfois assez douleureux, le logging aussi.	Le style REST + PubSub, ça reste assez facile avec les bons outils.
L'asynchronicité est obligatoire.	Il existe cependant un bridge HTTP.	Le style REST n'est pas nécessairement asynchrone.

Les déavantages (2)

Argument	WAMP	REST API + PubSub
La documentation n'est jamais réellement à jour...	Never trust the documentation. Trust the source code.	Les articles sur REST et les tutos sur Redis, et compagnie, y'en a des milliers.
Dépendances	Un routeur en Python (Crossbar)	Un système de PubSub (Redis, ZMQ, RabbitMQ, etc..)
Caching	Y'a plus de cache ! Il faut le faire soi-même manuellement, avec Redis par exemple.	Les requêtes HTTP peuvent être cachés car stateless.

Les déavantages (3)

Argument	WAMP	REST API + PubSub
Le style	Chacun ses goûts, mais le RPC n'est peut-être pas pour tout le monde.	Et vice-versa, le REST n'est pas pour tout le monde.

Le mot de la fin

Le realtime est difficile, mais cela peut devenir très plaisant avec les bons outils.

Voilà !

Vous apprécierez sûrement plus de ressources sur le sujet:

- SwarmJS (Offline + Online + Realtime): http://swarmjs.github.io/articles/todomvc/

Questions ?