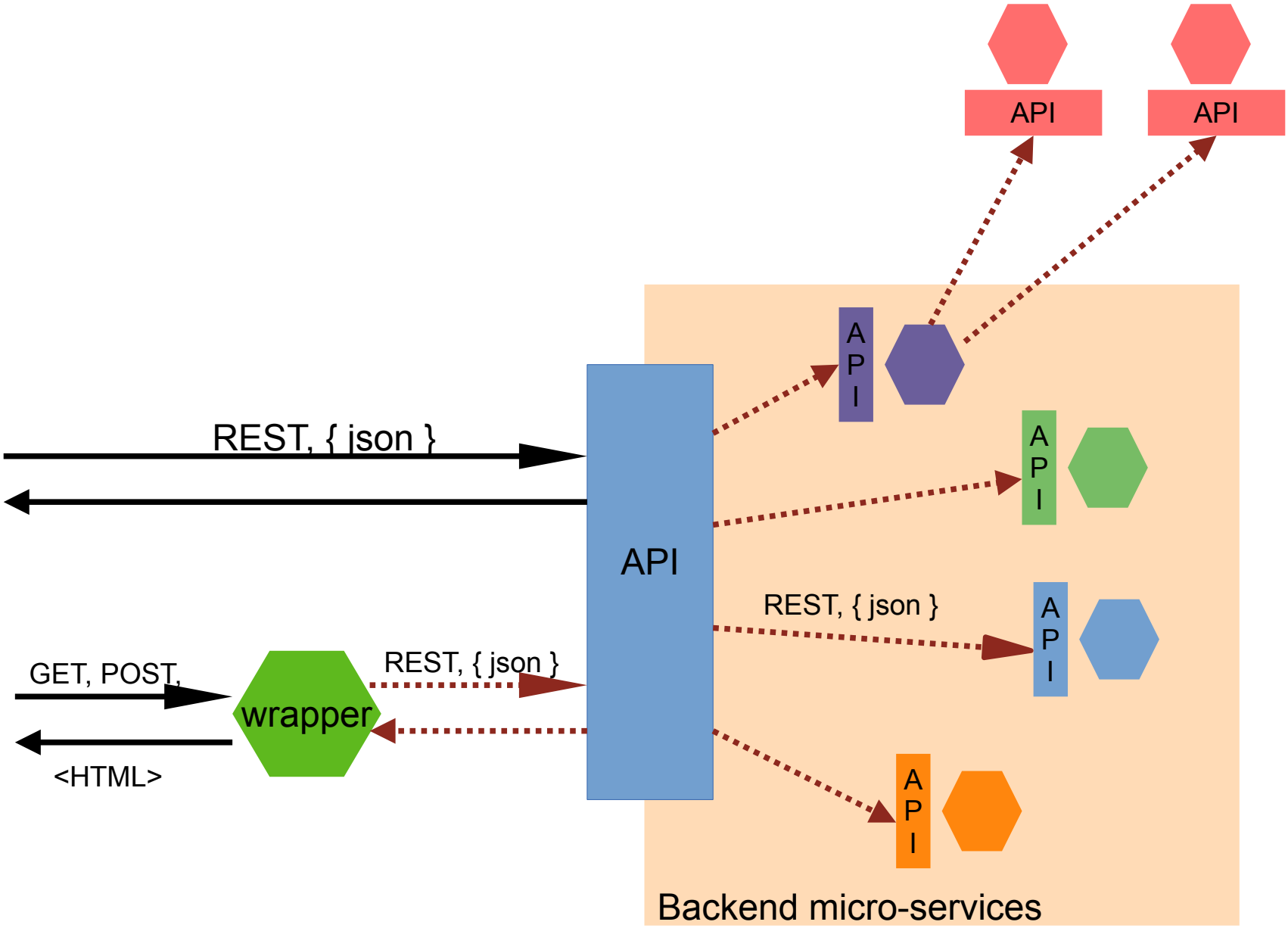
programmer un client REST

- Pour développer une application utilisant un service RestFul
- Pour tester une api

- **Programmer un client** :
  - Ligne de commande : curl
  - Php : file_get_content(), php curl
  - Utiliser une classe ou un micro-framework php
    - **Guzzle**

Clients REST côté serveur



curl en ligne de commande

```
$ curl -i -H "Accept: application/json" http://api.racoin.local/annonces/2

HTTP/1.1 200 OK
Date: Mon, 25 Jan 2016 14:30:21 GMT
Server: Apache/2.4.16 (Ubuntu)
Connection: close
Transfer-Encoding: chunked
Content-Type: application/json;charset=utf-8

{"annonce":{"id":2,"titre":"Sapiente ut fugiat sed quibusdam ea
voluptate.","descriptif":"Doloribus dolor animi quo excepturi. Vel exercitationem
exercitationem sunt qui ratione totam. Minima laboriosam nihil velit
est.","ville":"Garcia","code_postal":"67922","prix":"770.00","date_online":null,"
status":1,"cat_id":2,"type":"Annonce","categorie":{"id":2,"libelle":"v\
u00e9hicules\/moto","descriptif":"toutes les motos, 2 roues \u00e0
moteur"}},"links":{"annonceur":{"href":"\/annonces\/2\/annonceur"},"categorie":
{"href":"\/annonces\/2\/categorie"}}}
```

php curl

```php
// 1. initialisation
$ch = curl_init();

// 2. configutation de la requête
curl_setopt($ch,CURLOPT_URL,"http://www.nettuts.com");
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($ch, CURLOPT_HEADER, 1);

// 3. exécution et récupération du résultat
$output = curl_exec($ch);
$info = curl_getinfo($ch) ;
// 4. fermer
curl_close($ch);
```

Client Rest avec Guzzle

- Forge les requêtes avec la méthode choisie
- Ajoute des headers
- Envoie
- Récupère et décode les données de retour
- Sait parler xml et json

- installation :

```
{
    "require": {
        "guzzlehttp/guzzle": "^7.0"
    }
}
```

guzzle overview

créer un client :

```
use GuzzleHttp\Client;
$client = new Client([
    // Base URL : pour ensuite transmettre des requêtes relatives
    'base_url' => 'http://api.auth.local',
    // options par défaut pour les requêtes
    'timeout'  => 2.0,
]);
```

envoyer des requêtes :

```
$response = $client->get('/check');
$response = $client->get('http://api.fabrication.local/commandes');
$response = $client->post('/auth');
$response = $client->request('GET', '/check');
$r = $client→new Request( 'PUT', '/commandes/765EA-BE432')
$response = $client->send( $r ) ;
```

options pour les requêtes :

```
$response = $client→get('/commandes' , [
    'query' => ['s'=> 1, 'page'=>5] ,
    'headers' => [ 'Authorization'=> 'bearer gt65H8654',
                   'Content-Type' => 'application/json'
                 ] ] );
```

- Utilisation des réponses :

```php
$code = $response->getStatusCode();                    // 200
$contentType = $response->getHeader('Content-Type');
$body = $response->getBody() ;
$json = json_decode($response->getBody()) ;
```

guzzle overview : upload

données brutes :

```
$r = $client->post( 'http://api.racoi.local.annonces', [
                'body' => 'titre=vélo%20rouge&description=un%20...'
]);
```

application/x-www-form-urlencoded

```
$response = $client->post( '/annonces', [
    'body' => [
        'titre' => 'vélo rouge',
        'description' => 'un beau vélo rouge et bleu',
        'prix' => 120.00
    ]
]);
```

application/json

```
$r = $client->put('/annonces/2/annonceur', [
    'json' => ['mail_a' => 'joe@bar.fr', 'tel_a'=>'0789776544']
]);
```

Exemple : transférer des requêtes d'une api vers une autre en Slim

```php
$client = new Client([
    'base_uri' => $this->c->get('settings')['auth_service'],
    'timeout'  => 5.0
]);

$response = $client->request('POST', '/auth', [
      'headers'=> ['Authorization' => $rq->getHeader('Authorization')]
    ]
);

return $rs->withStatus($response->getStatusCode())
    ->withHeader('Content-Type', $response->getHeader('Content-Type'))
    ->withBody($response->getBody());
```

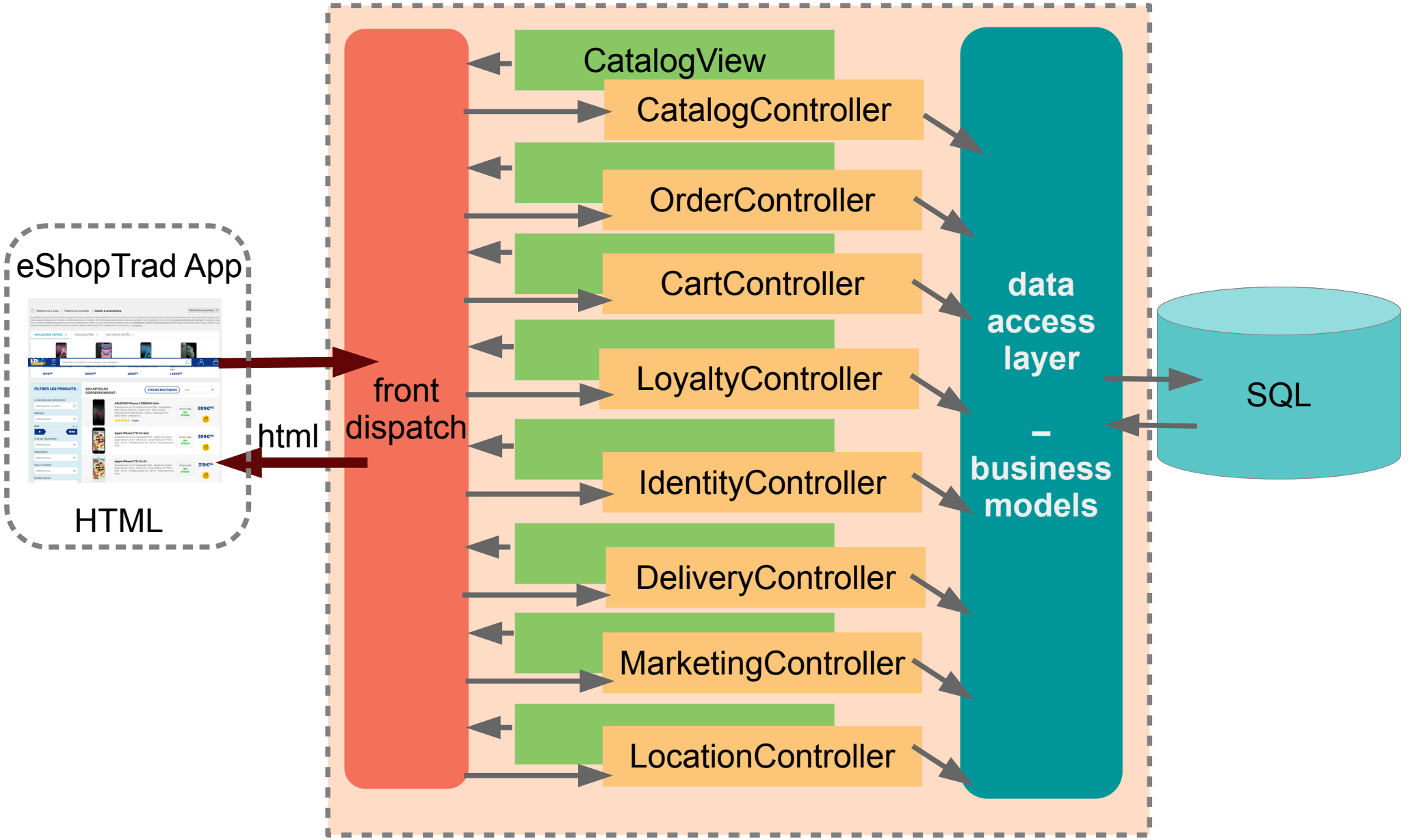
```php
$query= $rq->getQueryParams();
$response = $client->request('GET', '/commands', [
    'query'=>$query ] ) ;
```

Architecture micro-services

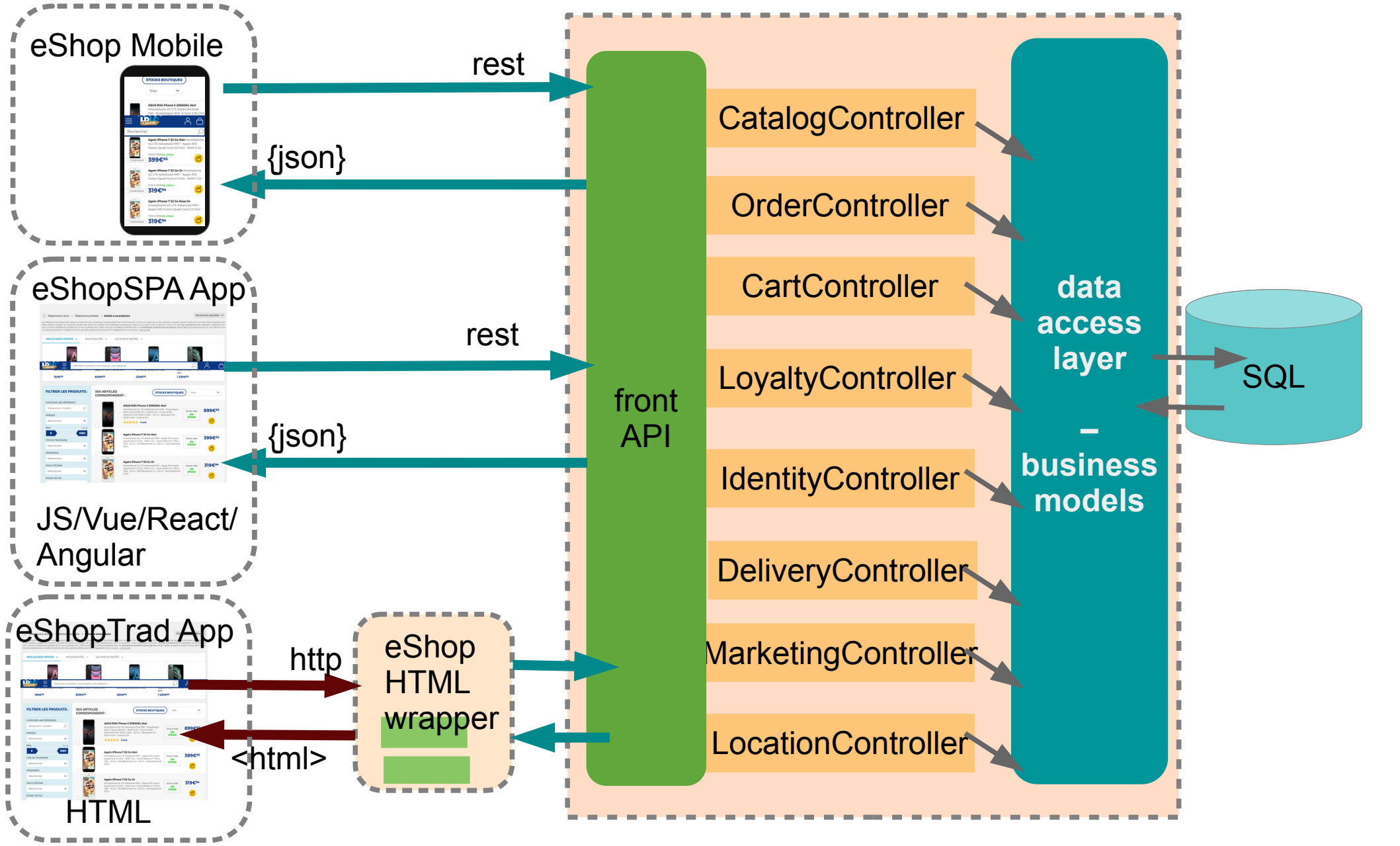
Les principes :

- le backend de l'application est décomposé en un ensemble de services de petite taille faiblement couplés, indépendants les uns des autres
- chaque service est autonome : il dispose de sa codebase propre et de ses données persistantes
- chaque service expose une api, et ne communique avec les autres services que au travers de leurs api

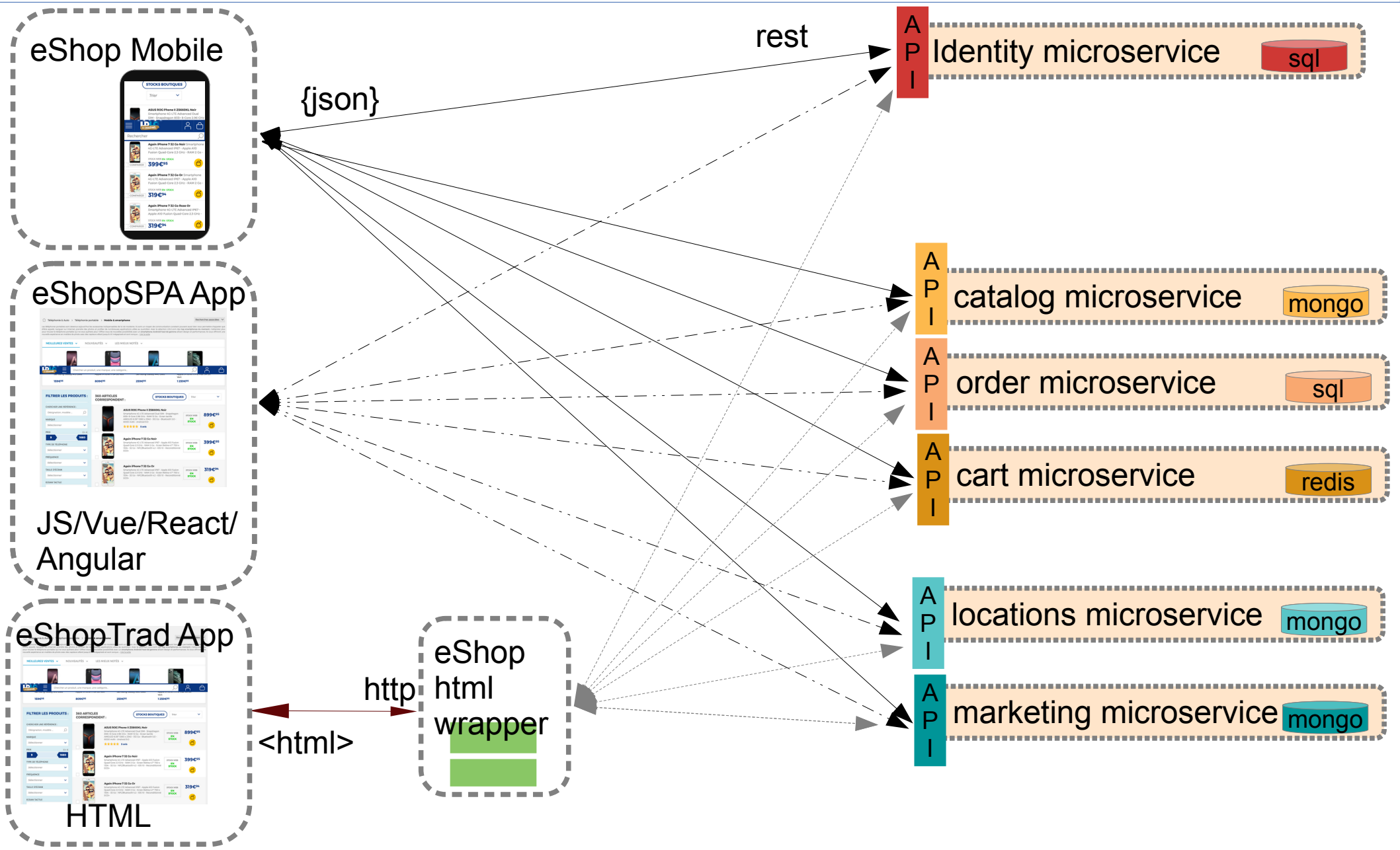
Exemple : eShop / architecture traditionnelle



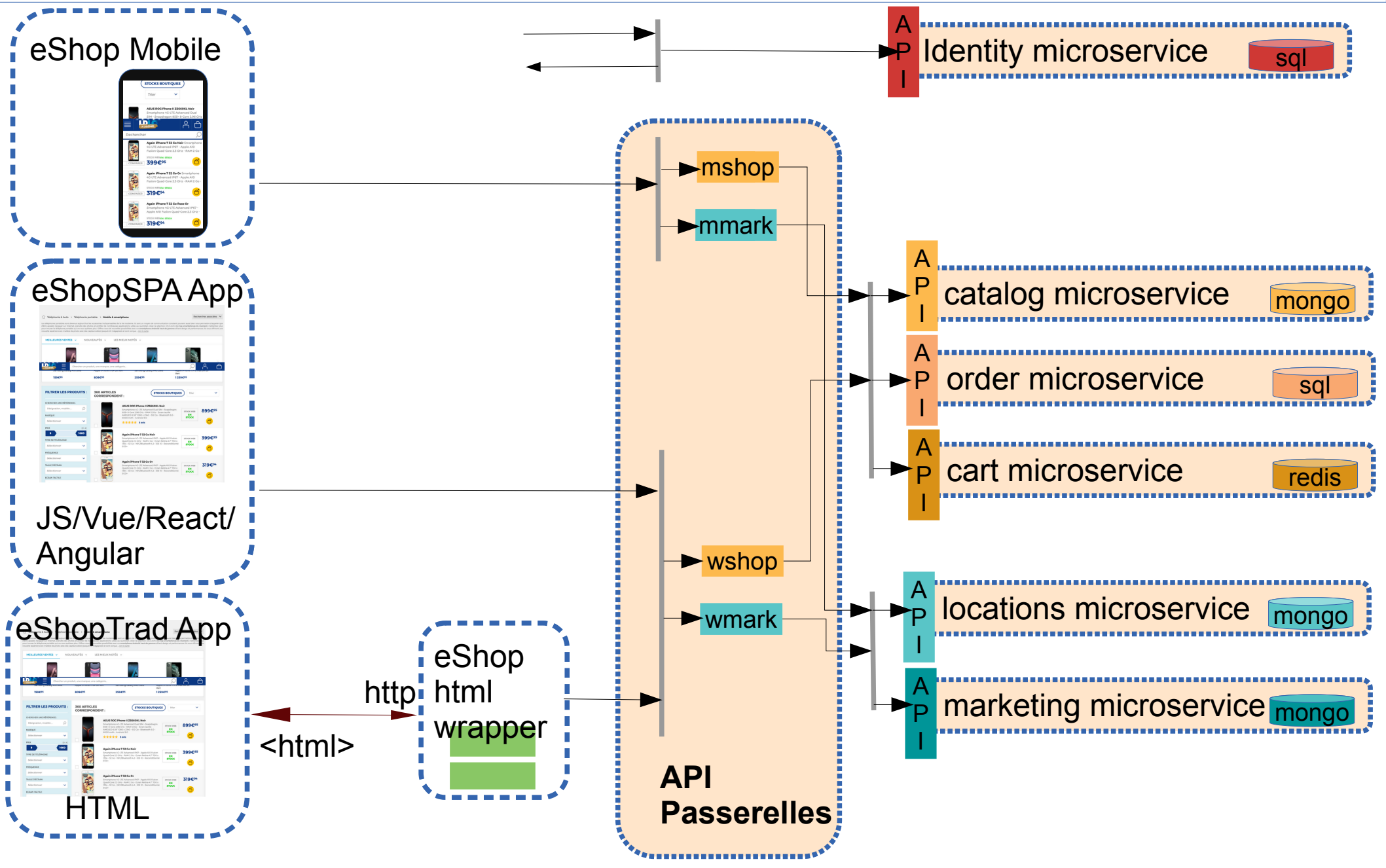
Exemple : eShop / architecture Single Page Application (SPA)



Exemple : eShop / architecture micro-services



Exemple : eShop / architecture micro-services



Architecture micro-service

Les avantages :

- des services de petite taille sont plus faciles à maintenir et faire évoluer ; on peut ajouter de nouveaux services
- développés, testés, déployés indépendamment les uns des autres par des équipes différentes
- technologies variées adaptées à chaque besoin
- services dupliqués facilement pour mieux passer à l'échelle

infrastructure

- développement : **docker-compose** permet de déployer les micro-services dans un ensemble de conteneurs docker **sur une seule machine**,

- production :
  - docker-compose pour de petites applications
  - Pour gérer la montée en charge : **kubernetes** ou **docker swarm** permettent de gérer la répartition des micro-services **sur plusieurs machines**