**Projet Fort Knox**



Prérequis :

* Avoir un serveur web local (Wamp, Lamp, ou Mamp),
* Un éditeur de texte ou un IDE (Sublime Text, NetBeans, IntelliJ).

Objectif :

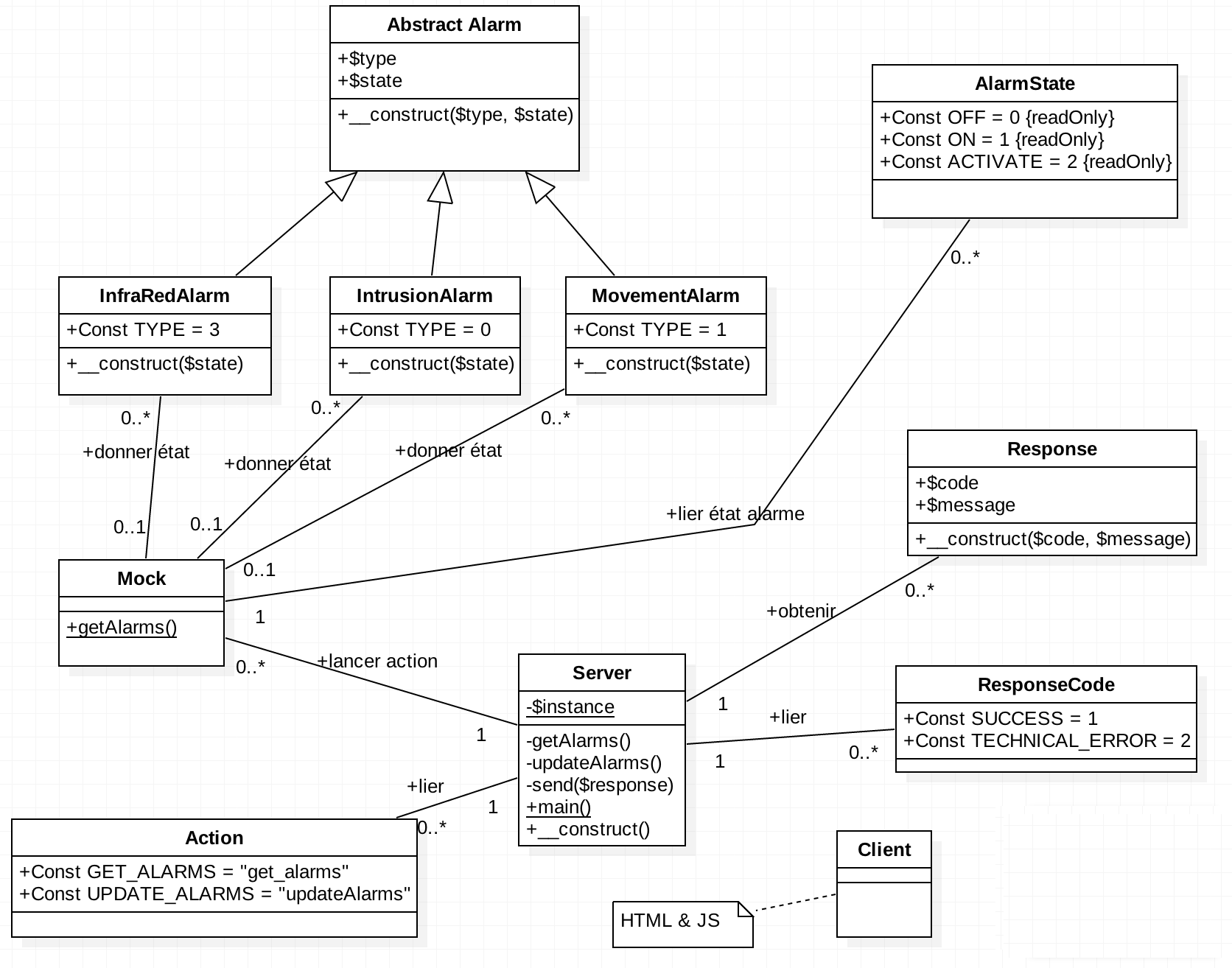
Concevoir une application web, client/serveur, qui gère le système d’alarme d’une maison.

Au travers de ce TP :

* Coté client, nous allons aborder l’utilisation de la librairie JavaScript jQuery, la conception d’une IHM, la communication avec un serveur web, et l’échange de données en JSON,
* Coté serveur, nous allons parler programmation PHP, POO, Classes, Héritage, Interfaces, types, et communication avec une base de données.

**NB :** ***IHM*** *signifie* ***I****nterface* ***H****omme* ***M****achine*. C’est la partie cliente (front) d’un site utilisant les compétences HTML, CSS, JS, jQuery, Ajax, etc. On parle aussi de maquette (ou phase de maquettage) en ergonomie (vu en S2) comme en conception (à voir en S3).

Soit le diagramme de classes suivant :



Maintenant que nous avons « codé » l’IHM de notre application, il nous reste à la connecter au serveur. Ce dernier est responsable du stockage des données, et expose des services pour les manipuler. Contrairement à notre IHM qui s’exécute dans le navigateur de l’internaute, le serveur « tourne » sur une machine dédiée et accessible par tous les navigateurs.

3 – Modifier la classe Server.php : en singleton

Cette classe sert de point d'entrée à notre serveur. C'est elle qui va recevoir les demandes du navigateur, les traiter, et s'assurer de la cohérence des réponses. C'est donc sur cette classe que notre IHM va se connecter et récupérer les données en JSON.

*Un élément de cours : un singleton est un patron de conception ou Design Pattern. L'objectif d'un singleton est de restreindre l'instanciation d'une classe à un seul objet (ou quelques uns* – *dans tous les cas, le nombre est très restreint).*

*Le singleton est utilisé lorsque l'on a besoin d'exactement UN objet pour coordonner des (plusieurs) opérations dans un système.*

* Modifiez le fichier Server.php (suppression du contenu),
* Créez (remplacez son contenu) une classe Server par les éléments ci-dessous,
* Le constructeur de la classe est déclaré en « public ». Pour l’instant, il est vide,
* La classe est un singleton (accès au même objet d’une classe) :
  + C’est-à-dire, créez un attribut statique privé nommé « instance »,
  + Ainsi qu’une méthode statique publique nommée « *main()* »,
* Dans la méthode statique « *main()* », instanciez la classe (ou appelez le constructeur de la classe) Server à partir de l’attribut statique « instance »,

*Rappel de la syntaxe : nomObjet = new NomClasse();*

*Attention l’attribut est statique, l’instanciation se fait donc à partir du mot du langage « self::… »… puisqu’on est dans la classe.*

* Par convention, créez cette méthode à la fin de la classe Server,
* Une fois la classe créée, invoquez la méthode statique *main()*. On dit qu’elle est auto-appelante : *Server::main();*
* Attention cette invocation doit être à l’extérieur de la classe.

4 – Créer la classe (de constantes) Action.php :

Il est impératif en programmation d’éviter de travailler avec des valeurs. Si la valeur vient à changer, cela peut générer des erreurs – des résultats erronés. Les classes de constantes et les énumérations (Enum) servent à faire abstraction des valeurs.

La classe Action représente la liste des demandes supportées par notre serveur, getAlarms pour la récupération des alarmes et updateAlarms pour la mise à jour.

* Créez un nouveau dossier **net**,
* A l’intérieur du dit dossier, créez la classe Action.php,
* La classe contient deux constantes : GET\_ALARMS et UPDATE\_ALARMS, ayant pour valeur respective “getAlarms” et “updateAlarms”.

5 – La classe Response.php (standardisation des retours) :

Cette classe est responsable du format que l'on va donner aux réponses du serveur. Il est important d'offrir un mécanisme uniformisé pour faciliter les échanges entre le serveur et le client (l’IHM). Dans notre cas, il s'agit d'un objet avec deux valeurs : code va indiquer le statut de la réponse, et message contient le corps de celle-ci.

* Dans le dossier **net**, créez cette fois la classe Response.php,
* Cette classe contient deux attributs : également *code* et *message*. Contrairement à l’habitude, ces attributs sont publics. Les attributs *publics* sont nécessaires pour la sérialisation en JSON,
* Le constructeur de la classe prend deux paramètres en entrée : également *code* et *message*,
* Dans le constructeur, utilisez ses paramètres pour affecter les valeurs des deux paramètres aux attributs de la classe.

6 – La classe ResponseCode.php :

Comme la classe Action, cette classe est une classe de constantes. Elle contient les différents statuts de réponse possible.

* Dans le dossier **net**, créez une nouvelle classe ResponseCode.php,
* Cette classe contient deux constantes : SUCCESS et TECHNICAL\_ERROR, ayant pour valeur respective 1 et 2.

7 – La classe Alarm.php : les objets du model

Les classes du model sont utilisées pour structurer les données. Sans surprise, Alarm.php reprend la même structure que celle de notre JSON. Ici nous allons utiliser l'héritage pour typer nos alarmes. Un typage fort, même dynamique, garantit de ne pouvoir appliquer que des opérations légales aux données.

* Créez un nouveau dossier **model**,
* Dans le dossier **model**, créez la classe *abstraite* Alarm.php. Elle contient deux attributs publics : *type* et *state*. Les attributs sont, là aussi, publics pour la sérialisation en JSON,
* Le constructeur de la classe prend deux paramètres en entrée : *type* et *state*,
* Dans le constructeur, utilisez ses paramètres pour affecter les valeurs de ses deux paramètres aux attributs de la classe,
* Dans le même dossier, créez les 3 sous-classes (ou classes concrètes) suivantes :
  + IntrusionAlarm.php,
  + MovementAlarm.php, et,
  + InfraredAlarm.php.
* Ces 3 classes héritent de la super classe *abstraite* Alarm.php,
* Pour ces trois sous-classes :
  + A chaque déclaration, utilisez *include\_once* afin de charger la super classe abstraite,

*Rappel de la syntaxe :*

*include\_once (\_\_DIR\_\_.’/chemin/nomClasse.php’);*

 Instruction à écrire avant la déclaration de la sous-classe.

* + Ces sous-classes disposent d’une constante :
    - Pour la classe IntrusionAlarm, la constante TYPE = 0,
    - Pour la classe MovementAlarm, la constante TYPE = 1,
    - Pour la classe InfraredAlarm, la constante TYPE = 3,
  + Chaque constructeur de chaque sous-classe appelle le constructeur de la super classe avec en paramètre *state* issu de la super classe. Le paramètre *type* est en fait une constante contenue dans la sous-classe dont la valeur de type est différente en fonction de la sous-classe. Il n’est donc pas passé en paramètre.

Attention à la notation du constructeur d’une sous-classe, l’instanciation se fait à partir du mot du langage « *parent::…*».

8 – La classe AlarmState.php :

Cette nouvelle classe de constantes est nécessaire pour connaître les valeurs que peut prendre l’attribut state d'une Alarm.

* Dans le dossier **model**, créez une nouvelle classe AlarmState.php,
* Cette classe contient trois constantes : OFF, ON, ACTIVATE, ayant pour valeur respective 0, 1 et 2.

9 – La classe Mock.php : le bouchon

A ce niveau, nous allons créer une classe de services. Elle permet le dialogue avec le navigateur (l’IHM ou encore la vue) et un modèle de données.

Il manque l’étape finale à ce processus : un jeu de données dynamique qui permet l’interaction entre les données manipulées par l’internaute issues de la base de données.

Un *bouchon* est utilisé pour fournir ce jeu de données. Cette étape intermédiaire permet de tester une application avec des valeurs « forcées ». Dès l’application valide et validée, l’étape suivante consiste à remplacer le bouchon par la connexion directe avec la base de données.

* Dans le dossier **model**, créez une classe Mock.php,
* La classe contient une méthode statique *getAlarms()*. Cette méthode retourne une liste d’Alarmes (résultat retourné : un tableau),
* Pour ce faire, utilisez les trois types d’alarme :
  + La classe IntrusionAlarm.php,
  + La classe MovementAlarm.php, et,
  + La classe InfraredAlarm.php et,
* La classe AlarmState permet de définir leur état,
* Pour ce faire, ajoutez les trois sous-classes ci-dessus avant la déclaration de la clas-se Mock :
  + Utilisez *include\_once* afin de charger les classes à utiliser ;
* Ajoutez également la classe de constantes AlarmState.php à partir de l’instruction *include\_once*,
* Comment écrire cette méthode statique :
  + Déclarez une variable *result* de type tableau,
  + Instanciez les classes concrètes à partir des constantes de la classe AlarmState.php :

- En premier IntrusionAlarm(), le paramètre est à ON

- puis MovementAlarm(), le paramètre est à OFF

- puis InfraRedAlarm(), le paramètre est à ON

- puis InfraRedAlarm(), le paramètre est à ON

- puis IntrusionAlarm(), le paramètre est à ACTIVATE

* + Chaque instanciation de la classe concrète est une « affectation » dans le tableau. N’oubliez pas : le constructeur possède un paramètre défini à partir de la classe de constantes AlarmState,
  + N’oubliez pas de retourner le résultat (*result*).

10 – Le traitement des demandes :

Il est temps maintenant de traiter les demandes qui seront envoyées à Server.php. Nous avons prévu de faire des appels en http via Ajax depuis notre IHM. Pour faire simple, nous passerons nos paramètres en GET.

Rappel : les paramètres en GET sont passés en clair dans l’url par opposition aux paramètres passés en POST « cachés ». Cf. TP sur le formulaire au semestre 1.

*Exemple GET :* <http://localhost/Server.php?variable1=valeur1&variable2=valeur2>

* Nous allons écrire maintenant le constructeur de la classe Server.php,
* Il contient une variable locale *action*. La valeur du paramètre *action* est passée ici en GET : *$\_GET['variable']*. Cette valeur provient d’un formulaire, d’une varia-ble…
* A l’aide d’un choix multiple, testez la variable *action* à partir des constantes de la classe Action.php,
* Pour chaque cas :
  + Pour la constante GET\_ALARMS : invoquez la méthode *getAlarms()*,
  + Pour la constante UPDATE\_ALARMS : invoquez la méthode *updateAlarms()*et,
  + Dans tous les autres cas : invoquez la méthode *sendUnknownActionError()*,
* Écrire le squelette des trois méthodes privées ci-dessus appelées dans le constructeur :
  + Méthode *updateAlarms()* – sans paramètre,
  + Méthode *getAlarms()* – sans paramètre,
  + Méthode *sendUnknownActionError()* – sans paramètre,
* Écrire une méthode d’envoi du jeu de données, pour l’instant, vide :
  + Méthode *send()* – avec le paramètre : *response*.

11 – Les retours :

Maintenant que nous traitons les demandes, il nous faut récupérer le model et retourner la réponse. Comme il est important de spécialiser les traitements, la récupération du model se fait par la méthode getAlarms() et les réponses sont toutes gérées par la méthode send().

* Nous allons compléter la classe Server.php,
* La méthode *send()* :
  + Elle prend en entrée le paramètre *response*,
  + Affichez le paramètre *response* converti en JSON à partir de la méthode *json\_encode(…)*,
* La méthode *getAlarms()* :
  + Récupérez le bouchon défini dans la classe Mock pour la méthode *getAlarms()*, dans une variable locale *alarms* : *Mock::getAlarms();*
  + Créez une instance de la classe Response en succès. Pensez à utiliser pour ce faire des constantes de la classe ResponseCode,
  + Utilisez votre liste d'alarme comme valeur de l’attribut message de l'instance de la classe Response,
  + Invoquez la méthode *send()* en lui passant en paramètre l'instance de la classe Response.
* La méthode *updateAlarms()* :
  + Est vide pour l’instant.
* La méthode *sendUnknownActionError()* :
  + Créez une instance de la classe Response en erreur. Pensez à utiliser pour ce faire des constantes de la classe ResponseCode,
  + Utilisez une chaine de caractères 'action inconnue' comme valeur de l’attribut message de l'instance de Response.
  + Invoquez la méthode *send()* en lui passant en paramètre l'instance de la classe Response.

12 – Le test :

Tout est prêt pour câbler l’IHM avec notre serveur. Mais avant il convient de tester si tout fonctionne correctement.

* **Pour obtenir notre JSON :**
* Affichez la page Server.php en passant *getAlarms* dans la variable locale *action* en GET,

Ce qui donne en final d’url : [Server.php?action=getAlarms](http://localhost/fortknox/Server.php?action=getAlarms)

Et pour une arborescence simple (serveur Apache + fortknox) sur le serveur :

<http://localhost/fortknox/Server.php?action=getAlarms>

Ce qui donne en résultat, le JSON :



* **Pour obtenir un message d'erreur :**
* Affichez la page Server.php en passant *makeCoffee* dans la variable locale *action* en GET,

Ce qui donne en final d’url : [Server.php?action=makeCoffee](http://localhost/fortknox/Server.php?action=getAlarms)

Et pour une arborescence simple (serveur Apache + fortknox) sur le serveur :

<http://localhost/fortknox/Server.php?action=makeCoffee>

Ce qui donne en résultat, le JSON :



Attention sur votre serveur Apache et selon votre arborescence, vous pouvez avoir des différences (localhost : Sous MAC s’écrit localhost:8888 – Sous Windows quand tout va bien : localhost, sinon : localhost:8080 +++ les sous-dossiers ajoutés.

Par exemple, concernant mon serveur Apache, c’est plutôt du style (;-)) :

<http://localhost:8888/2015_2016_M2202(TP)/Semestre_2/TP03_Corr/fortknox/Server.php?action=getAlarms>

13 – Le proxy : on « dé-bouchonne »

A ce stade, notre client JavaScript communique avec notre serveur en PHP. Il est temps de retirer notre bouchon et d'utiliser de véritables données. Un proxy est une classe dont le rôle est d’échanger avec un fournisseur de données. Cela peut être un Web Service, un fichier ou un bouchon, etc.

* Dans le dossier **net**, créez une nouvelle classe AlarmProxy qui contient :
  + Une constante,
  + Une méthode publique de type « fonction »,
* Dans la classe AlarmProxy :
  + Créez une constante FILE\_PATH ayant pour valeur 'data.json',
  + Créez une méthode publique getAlarms() – sans paramètre,
* Dans cette méthode *getAlarms()* :
  + Vous aurez besoin de 2 variables locales : data et *json*
  + Chargez le contenu du fichier 'data.json' dans une variable locale data à partir de l’instruction *file\_get\_contents(…)*.

La syntaxe est la suivante : *file\_get\_contents('monfichier.ext');*

Attention, le fichier à passer en paramètre est celui de la constante FILE\_PATH

Pour plus d’informations sur l’instruction : Cf. le site php.net à l’adresse ci-dessous

<http://php.net/manual/fr/function.file-get-contents.php>

* + Comme nous avions encodé en JSON précédemment les données, nous allons maintenant les décoder ou les « désérialiser »,
  + Pour cela, nous allons stocker ces données ci-dessus (data) à partir de la méthode *json\_decode(…)*dans la variable locale *json*,

La syntaxe est de la forme : variable = *json\_decode(…);*

*En informatique, la sérialisation est un processus visant à coder l'état d'une information sous la forme d'une suite d'informations plus petites…*

* + La méthode étant une fonction, elle retourne la variable *json*.

14 – Nos données : data.json

Notre proxy va essayer de lire un fichier qui n'existe pas, il nous faut donc le créer.

* Dans le même dossier (**fortknox**) que la classe Server.php, créez un fichier de type texte nommé data.json,
* Ajoutez dans ce fichier le bouchon JSON vu au TP1. Ce fichier représente la liste des Alarmes.

De la forme :

*[{"type":0,"state":1},{"type":1,"state":0},{"type":3,"state":1},{"type":3,"state":1},{"type":0,"state":2}]*

* Testez la classe AlarmProxy.php dans le navigateur afin de s’assurer qu'il n'y a pas d'erreur, comme vu précédemment,
* Pour ce faire, dans la classe Server.php, remplacez l'appel à Mock::getAlarms() dans la méthode getAlarms() par le proxy – c’est-à-dire les 3 étapes ci-dessous :
* Etape 1 : Ajoutez un nouvel attribut privé proxy,
* Etape 2 : Au début du constructeur, instanciez la classe AlarmProxy.php dans l’attribut proxy,
* Etape 3 : Dans la méthode getAlarms(), remplacez l'affectation de la variable alarms avec Mock::getAlarms() par $this->proxy->getAlarms().

15 – Administration

Nous avons une IHM qui affiche les alarmes fournies par le serveur. Maintenant nous allons créer une nouvelle IHM pour les administrer. Le script *admin.php* est une IHM au même titre que le script *client.php* du TP01, seul le rôle diffère.

**Un élément de cours :** ***IHM*** *signifie* ***I****nterface* ***H****omme* ***M****achine*. C’est la partie cliente (front) d’un site utilisant les compétences HTML, CSS, JS, jQuery, Ajax, etc. On parle aussi de maquette (ou phase de maquettage) en ergonomie (vu en S2) comme en conception (à voir en S3).

* Dans le même dossier (**fortknox**) que la classe client.php, créez un nouveau script admin.php,
* Dans un premier temps, le script est vide. Il s’agit d’une simple page html, comme celle du script client.php. Créez une structure html basique,

Par exemple :

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8">  <title>Admin</title>  <script> </script>  <style>  </style>  </head>  <body>  </body>  </html> |

* Dans le dossier **js**, créez un nouveau fichier *admin.js*. Là aussi le script js est vide,
* Dans le script admin.php, ajoutez les scripts js permettant l’utilisation de la bibliothè-que jQuery. A savoir :
* D’abord le script jQuery :

<script src="js/jquery-2.2.1.min.js"></script>

* Puis le script admin.js :

<script src="js/admin.js"></script>

Cf. la syntaxe exacte dans le script client.php.

16 – Le formulaire dans admin.php

L'idée est d'utiliser un formulaire afin d’afficher et modifier l'état de nos alarmes.

* Le formulaire est à créer dans le script admin.php,
* Dans la balise <body> </body>, créez une balise formulaire : <form> </form>

Cf. le formulaire fait au semestre 1 ou par exemple le site du W3schools :

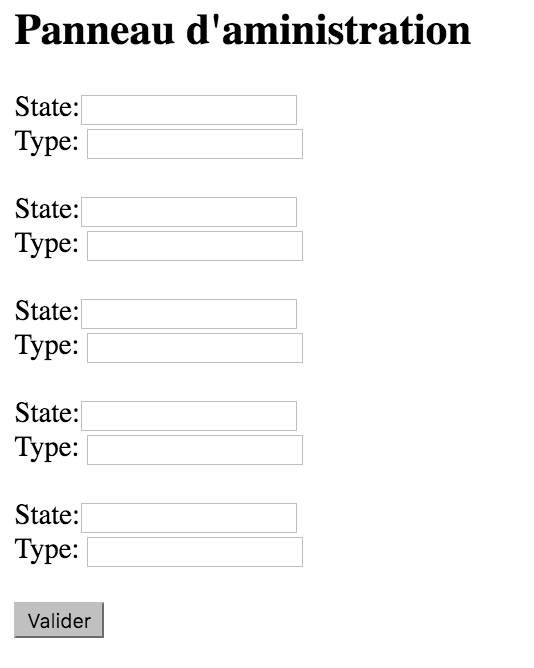
<http://www.w3schools.com/html/html_forms.asp>

* Ajoutez dans la balise <form> une balise <div> ayant pour id alarm0,
* Dans votre balise <div> ajoutez deux balises <input> de type text, une ayant pour nom alarmState et une autre ayant pour nom alarmType,
* Dupliquez votre balise <div> autant de fois que vous avez d'alarmes et incrémentez l'id correspondant,

Exemple : id='alarm1', id='alarm2' puis id='alarm3', etc.

* Enfin, ajoutez une balise <button> APRES la balise de fermeture du formulaire </form>, avec pour id la valeur sendButton,

Le formulaire doit être de la forme :



* Puis testez votre page dans le navigateur.

17 – Le script admin.js

Il s'agit du script admin.js qui « va » piloter le comportement de l’IHM admin.php, comme nous l’avons fait aux TP1 et TP2 quand le script fortknox.js pilotait le script client.php.

* Récupérez du script fortknox.js : les méthodes getAlarmsAjax(), updateAlarm(), et $(document).ready

Puisque nos deux IHM (client.php & admin.php) reposent sur les mêmes *services*, il est normal d'avoir du code en commun…

* Videz le contenu de la fonction anonyme passée en paramètre de (document).ready et appelez getAlarmsAjax() dedans,

Au chargement de la page, le JS va appeler la méthode getAlarmsAjax() afin de récupérer la liste des alarmes, comme nous l’avons fait au TP2 pour client.php.

* Modifiez la méthode updateAlarm() :

La principale différence réside ici puisque cette méthode est responsable de l'affichage.

Dans la boucle **for,** utilisez la bibliothèque jQuery afin de modifier la valeur des balises <input> : https://learn.jquery.com/using-jquery-core/selecting-elements/

S*yntaxe : Utilisez find('input') après le sélecteur JQuery afin d’obtenir la liste des balises <input> de la balise <div>.*

* Puis testez votre page admin.php.

18 – Mise à jour des données

Si notre IHM permet de modifier les données, il faut encore les récupérer pour les envoyer au serveur.

* Dans admin.js, créer une nouvelle fonction sans paramètre *sendAlarms()*
* Dans cette fonction, utiliser *console.log* pour afficher le message 'sending alarms…'  
  <https://developer.mozilla.org/fr/docs/Web/API/Console/log>
* Dans admin.php, utiliser l'attribut *onclick* de l’élément **sendButton**, pour appeler la fonction *sendAlarms()*[*https://developer.mozilla.org/fr/docs/Web/API/Console/log*](https://developer.mozilla.org/fr/docs/Web/API/Console/log)
* Tester votre page, au clic sur le bouton un message doit apparaître dans la console.  
    
  *Rappel : la console est disponible dans les outils de développeurs de votre navigateur.*
* Nous allons maintenant récupérer les informations du formulaire. A la fin de la fonction *updateAlarm()*, utiliser l'opérateur **this** pour sauvegarder la variable locale **alarms** dans une autre du même nom mais dans un contexte global (ouf)  
  [https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Op%C3%A9rateurs/L\_op%C3%A9rateur\_this](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Opérateurs/L_opérateur_this)  
  De cette façon nous conservons la liste des alarmes pour être utiliser ultérieurement.
* Dans la fonction *sendAlarms*, créer une boucle for de 0 à **this.alarms.length**
* Dans la boucle, utiliser JQuery comme dans *updateAlarm*, pour parcourir le formulaire et mettre à jour les alarmes de **this.alarms**.
* Il ne nous reste plus qu'à envoyer les données au serveur. Pour cela nous allons utiliser Jquery pour faire un appel Ajax comme dans la fonction *getAlarmsAjax*.
  + Configurer le **type** POST. Pour comprendre la différence entre POST et GET, consultez cette page : <http://www.w3schools.com/tags/ref_httpmethods.asp>
  + Configurer l'**url** [http://localhost/fortknox/Server.php?action=updateAlarms](http://localhost/fortknox/Server.php?action=getAlarms)  
    Notez que nous avons changé l'**action** par updateAlarms
  + Configurer **data** en lui affectant *this.alarms* encodé en JSON  
    <https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/JSON/stringify>
  + Configurer **contentType** en lui affectant la valeur "application/json; charset=utf-8"  
    Pour info : contentType sert à expliquer au serveur qu'on lui envoie du JSON et dataType pour lui expliquer ce qu'on attend en retour.
  + Enfin, configurer **dataType** avec "json"
  + Comme dans la fonction *getAlarmsAjax*, nous allons utiliser la promesse **done** pour afficher la réponse du serveur en console.
  + Tester :)

19 – Mise à jour des données coté serveur

Notre IHM envoie les données au serveur qui n'en fait rien pour l'instant. Nous allons voir comment récupérer ses données et les sauvegarder dans le fichier **data.json**.

* Dans Server.php, le constructeur traite déjà la demande de mise à jour et invoque la méthode *updateAlarms()* Celle ci est normalement vide mais ça va changer.
* Créer une nouvelle variable local $data et lui affecter file\_get\_contents('php://input')

**php://input** est un flux en lecture seule qui permet de lire des données brutes depuis le corps de la requête. Consulter <http://php.net/manual/fr/wrappers.php.php>pour plus d'information.

* Désèrialiser $**data** et affecter cette valeur dans une nouvelle variable local $**dataJson**. Voir le chapitre 13 sur la sérialisation JSON.
* Ensuite, invoquer la méthode du proxy *updateAlarms()* en lui passant en paramètre $**dataJson**. Attention, cette méthode n'existe pas, nous allons la créer juste après. Pour rappel, il existe une instance de **AlarmProxy** dans une propriété de notre classe.
* Enfin, comme pour la méthode *getAlarms()*, utiliser la méthode *send()* pour retourner une **Response** en succès avec le message 'Update Complete'
* Dans la classe **AlarmProxy** créer une nouvelle méthode publique *updateAlarms()* qui prend un paramètre $alarms
* Dans cette méthode, créer une nouvelle variable locale $json et lui affecter $alarms sérialiser en JSON.
* Utiliser *file\_put\_contents* pour sauvegarder dans le fichier data.json. <http://php.net/manual/fr/function.file-put-contents.php>
* Tester la modification de donnée et voir si ça fonctionne en rafraichissant la page d'admin.
* Tester en rafraîchissant aussi la page client.php

~~- modif proxy php~~

- Exceptions

- Interfaces : nouveau type d'alarmes

- namespaces

- config server

- abstract factory

- bdd