

Documentation API

Table des Matières

[1 Présentation générale 2](#_Toc208571705)

[2 La gestion des modules 3](#_Toc208571706)

[2.1 Les modules : IModule 3](#_Toc208571707)

[2.2 Le gestionnaire de ressources : IRessourcesManager 3](#_Toc208571708)

[2.3 Gestion de points d’ancrage : IHookManager 4](#_Toc208571709)

[2.4 Chargement & déchargement des modules 5](#_Toc208571710)

[2.5 Les symboles : create et destroy 5](#_Toc208571711)

[3 Les outils 6](#_Toc208571712)

[3.1 L’interface principale : ITools 6](#_Toc208571713)

[3.2 Le message http : IMessage 6](#_Toc208571714)

[Figure 1 Modélisation du paramètre des modules 6](file:///C:\Users\Administrator\Desktop\DocumentationAPI.docx#_Toc208571715)

[3.2.1 La requête : IRequest 7](#_Toc208571716)

[3.2.2 La réponse : IResponse 8](#_Toc208571717)

[3.3 Le fichier de configuration : IConfig 8](#_Toc208571718)

[3.4 Les infos de connexion : IConnectionInfos 9](#_Toc208571719)

[4 Annexes 9](#_Toc208571720)

[4.1 Exemple : un module de test 9](#_Toc208571721)

# Présentation générale

Le but de cette API[[1]](#footnote-2) est de fournir un patron de conception, destiné aux développeurs désirant ajouter des fonctionnalités à notre serveur web par l’intermédiaire de modules. Elle implémente deux fonctions principales, d’une part assurer la compatibilité entre les différents modules et le serveur, d’autre part contrôler les ressources utilisées par les différents modules pour assurer la stabilité du serveur.

(CF le diagramme UML)

Notre API se décompose en deux parties distinctes, une première partie concernant la gestion des modules et une seconde concernant les outils disponibles.

# La gestion des modules

## Les modules : IModule

L’interface principale est la classe IModule. Le principe de fonctionnement des modules est le suivant : Les modules peuvent contenir jusqu'à 6 méthodes pouvant être appelées au sein de la chaine de traitement. Pour cela on utilise la méthode getCallback qui renvoie un tableau avec ces différentes méthodes.

Cette interface dispose de 3 méthodes :

|  |
| --- |
| class IModule : public virtual IRessourcesManager  {  public:  typedef void (IModule::\*p\_callback)(ITools&);  virtual ~IModule() {};  virtual std::pair<p\_callback, IHookManager::hookPosition>\* getCallbacks() = 0;  virtual bool onLoad() = 0;  virtual void onUnload() = 0;  }; |

* onLoad, une méthode appelée lors du chargement du module
* onUnload, une méthode appelée lors du déchargement du module
* getCallbacks, une méthode qui renvoie le tableau contenant la callback[[2]](#footnote-3) à exécuter pour chaque point de hook[[3]](#footnote-4).

## Le gestionnaire de ressources : IRessourcesManager

Cette classe est héritée par la classe IModule et fournie des surcharges d’appels système tels que malloc, free, new, delete, open ou encore close. La méthode clean du manager doit être appelée dans les destructeurs des classes héritant de IRessourcesManager. Elle a pour effets de désalouer la mémoire non libérée et de fermer les fichiers encore ouverts. Le manager de ressources permet ainsi d’assurer la stabilité du système.

|  |
| --- |
| class IRessourcesManager  {  public:  virtual ~IRessourcesManager() {};  virtual void\* rm\_alloc(size\_t size) = 0;  virtual void rm\_desalloc(void\* ptr) = 0;  virtual int rm\_open(const char\* pathname, int flags) = 0;  virtual int rm\_close(int fd) = 0;  virtual void rm\_clean() = 0;  }; |

## Gestion de points d’ancrage : IHookManager

Concernant l’appel des fonctions d’un module au sein du serveur notre API fournit une interface permettant de gérer les différents points d’ancrage.

|  |
| --- |
| class IHookManager  {  virtual bool addModule(std::pair<IModule::p\_callback,  hookPosition>[hookPointsNumber], IModule\*) = 0;  virtual bool removeModule(IModule\*) = 0;  virtual bool manageHookPoint(hookPoint, ITools&) = 0;  };  } |

Notre API définie 8 points d’ancrage différents :

* 4 points d’ancrage différents au sein de la chaine de traitement.
* 1 point d’ancrage à la connexion d’un client, **NEW\_CLIENT**.
* 1 point d’ancrage à la déconnexion d’un client, **DEL\_CLIENT**.
* 1 point d’ancrage à la lecture sur la socket, **READ.**
* 1 point d’ancrage à l’écriture sur la socket, **WRITE.**

Au sein d’un même point de hook, notre API définie 5 positions possibles par module FIRST, MIDDLE, LAST, VERY\_FIRST et VERY\_LAST. Il est possible d’avoir plusieurs modules dans les 3 premières positions au même point de hook, en revanche il ne doit y avoir qu’un seul module en VERY\_FIRST et un seul module en VERY\_LAST par point de hook. Ces deux dernières positions ne doivent être utilisées que dans de rare cas, par exemple un module de compression doit se trouver en VERY\_LAST car on ne doit plus modifier la réponse après l’avoir compressée.

Le hookManager devra également implémenter les 3 méthodes suivantes :

* addModule qui permet d’enregistrer une module au sein du hook manager. Cette méthode prend en paramètre, le tableau renvoyé par la méthode getCallbacks de IModule et un pointeur sur l’instance du module.
* removeModule qui permet de retirer un module du hook manager.
* manageHookPoint qui permet de gérer un point de hook donné. Cette méthode prend en paramètre le point de hook à manager ainsi que le ITools à passer aux différents modules.

## Chargement & déchargement des modules

Le serveur qui implémentera notre API se devra de pouvoir charger des décharger des modules à la volée. Il existe de nombreuses manières différentes de gérer cette problématique. C’est pourquoi nous avons décidé de ne pas imposer la manière dont les modules seront chargés au sein du serveur. Imposer la méthode pour gérer les points de hook nous semble suffisante pour assurer la compatibilité entre les modules.

## Les symboles : create et destroy

Nous sommes contraints d’imposer les noms des symboles permettant la création et la suppression des modules. Ils seront déclarés grâce au mot clé « extern ‘’C’’ » pour que le nom reste inchangé à la compilation. Nous utilisons deux symboles au sein de notre API, un symbole correspondant à la création du module et un module correspondant à la destruction.

|  |
| --- |
| extern "C" type\_du\_module \* create()  {  return new type\_du\_module();  }  extern "C" void destroy(void\* p)  {  delete (static\_cast< type\_du\_module \*>(p));  } |

# Les outils

## L’interface principale : ITools

Figure Modélisation du paramètre des modules

L’interface ITools représente l’unique paramètre passé aux callbacks. Elle est composée de trois méthodes d’accès sur IMessage, IConfig et IConnectionInfos dont les fonctionnalités respectives sont décrites dans la section suivante, ainsi que d’un membre prive, data, une std ::string utilise pour la connexion sécurisée.

## Le message http : IMessage

Le message http est décomposé en deux parties accessibles depuis l’interface IMessage. Cette interface est décomposée en deux sous-interfaces, IRequest et IResponse, qui implémentent des méthodes d’accès et de modification sur la requête et la réponse respectivement.

### La requête : IRequest

La requête contenant toute une série de méthodes permettant d’accéder et de modifier les éléments d’une requête.

|  |
| --- |
| class IRequest  {  public:  virtual ~IRequest() {};  virtual std::string& HTTPVersion() = 0;  virtual void HTTPVersion(const std::string& version) = 0;  virtual const std::string& method() = 0;  virtual const std::string& uri() = 0;  virtual const std::map<std::string, std::string>&  getHeaders(const std::string& field) = 0;  virtual void setHeaders(const std::string& field,  std::map<std::string, std::string>& headers) = 0;  virtual const std::string& getHeader(const std::string& field,  const std::string& key) = 0;  virtual void setHeader(const std::string& field,  const std::string& key,  const std::string& value) = 0;  virtual const std::string& getHeader(const std::string& header) = 0;  virtual void setHeader(const std::string& field,  const std::string& value) = 0;  virtual const std::string& body() = 0;  virtual void body(const std::string& body) = 0;  virtual void bodyAppend(const std::string& body) = 0;  virtual void bodyAppend(char\* body, uint size = 0) = 0;  virtual bool chunked() const = 0;  virtual void chunked(bool isChunked) = 0;  }; |

*Note : Les méthodes getHeader et getHeaders sont insensibles à la casse.*

### La réponse : IResponse

La réponse contenant toute une série de méthodes permettants d’accéder et de modifier les éléments d’une réponse ainsi qu’une méthode permettant de construire la réponse. Cette classe hérite de IRequest, elle dispose donc de tout les éléments cités à la page précédente.

|  |
| --- |
| class IResponse : public virtual IRequest  {  public:  virtual ~IResponse() {};  virtual void statusCode(int code) = 0;  virtual std::string& statusCode() = 0;  virtual void statusMessage(const std::string& message) = 0;  virtual std::string& statusMessage() = 0;  virtual std::string buildResponse() = 0;  virtual void resetHeaders() = 0;  }; |

## Le fichier de configuration : IConfig

Une interface permettant de dialoguer avec le fichier de configuration du serveur et permet également de savoir sur quel système d’exploitation est exécute le serveur.

|  |
| --- |
| class IConfig  {  public:  enum OS  {  LINUX,  WINDOWS,  OTHER  };  virtual ~IConfig() {};  virtual std::string& getParam(const std::string& param) = 0;  virtual int getParamInt(const std::string& param) = 0;  virtual OS getOS() = 0;  };  } |

## Les infos de connexion : IConnectionInfos

Cette classe n’est utile que lors de la gestion de connexion sécurisée. Elle contient la socket du client et un objet de type extension, par exemple de type SSL\*.

|  |
| --- |
| class IConnectionInfos  {  public:  typedef void Extension ;  enum connectionType  {  SIMPLE,  SSL  };  virtual ~IConnectionInfos() {};  virtual const SOCKET socket() = 0;  virtual const connectionType connectType() = 0;  virtual Extension\* extension() = 0;  virtual void extension(void\* ptr) = 0;  };  } |

# Annexes

## Exemple : un module de test

Pour illustrer le comportement de notre API, voici un exemple du module qui effectue 3 actions.

* Affichage d’un message au chargement du module
* Affichage d’un message au déchargement du module
* Enregistrement d’une callback auprès du hookManager au point d’ancrage PARSED en position LAST.

Test.hpp

|  |
| --- |
| #ifndef \_TEST\_H\_  # define \_TEST\_H\_  #include <iostream>  # include "IModule.hpp"  using namespace ZenZiAPI;  class ziamodule : public IModule  {  public:  bool onLoad();  void onUnload();  std::pair<p\_callback, hookPosition>\* getCallbacks();  void run(ITools&);  private:  std::pair<p\_callback, hookPosition> p\_int[hookPointsNumber];  };  #endif |

Test.cpp

|  |
| --- |
| #include "test.hpp"  using namespace ZenZiAPI;  bool ziamodule::onLoad()  {  std::cout << "LOADING…" << std::endl;  return (true);  }  void ziamodule::onUnLoad()  {  std::cout << "UNLOADING…" << std::endl;  }  /\* fonction principale de notre module \*/  void ziamodule::run(ITools&)  {  std::cout << "TEST MODULE IS RUNNING" << std::endl;  }  std::pair<IModule::p\_callback, IHookManager::hookPosition>\* ziamodule::getCallbacks()  {  /\* on enregistre notre methode run en tant que  Callback au point de hook PARSED \*/  this->p\_int[IHookManager::PARSED].first = static\_cast<IModule::p\_callback>(&ziamodule::run);  /\* on donne la priorite LAST à cette callback \*/  this->p\_int[IHookManager::PARSED].second = IHookManager:: LAST;  return p\_int;  }  /\* Déclaration du symbole create permettant d’acceder au constructeur  du module\*/  extern "C" ziamodule\* create()  {  return new ziamodule();  }  /\* Déclaration du symbole destroy permettant d’acceder au destructeur  du module\*/  extern "C" void destroy(void\* p)  {  delete (static\_cast<ziamodule\*>(p));  } |

1. API : *interface de programmation applicative*. [↑](#footnote-ref-2)
2. callback : *méthode du module appelée au sein du module, au niveau d’un point de hook.* [↑](#footnote-ref-3)
3. Point de hook: *point d’ancrage des modules dans la chaine de traitement*. [↑](#footnote-ref-4)