

EXPRESSO

AIDE AU DEVELOPPEMENT

EXPRESSO

« Aide au développement »

de Michael NashJcorporate Ltd.

Traduit de l'anglais par Yves AMAÏZO et Geneviève BOUDAUD

APERCU

Expresso est une application **framework** et une bibliothèque de composants.

En fait, son but est de servir de **framework** pour créer des systèmes d'application orientée WEB. C'est aussi une bibliothèque de composants au sens où vous pouvez utiliser ses composants sans utiliser Expresso comme votre **framework**

* par exemple, vous pouvez choisir d'utiliser Turbine (un autre **open source application framework**) comme **framework**, mais utiliser les objets base de données d'Expresso comme mécanisme de **persistance**.

Expresso dépend fortement d'au moins une base donnée disponible pour ses opérations, et convient le mieux aux applications pour lesquelles une base de données est une part essentielle.

Expresso fournit un certain nombre de services entrant dans les catégories générales suivantes :

INTRODUCTION

Bienvenue dans le guide de développement Espresso !

Ce guide a pour but de fournir au développeur java l'information nécessaire à l'utilisation de la bibliothèque de développement Espresso et/ou de développement d'applications Web avec le framework Espresso.

Il vous donne, sous une forme concise, le résultat net de plusieurs milliers d'heures de programmation avec Espresso et du développement des meilleures pratiques pour son utilisation et son implémentation.

Ce guide est le résultat de beaucoup de contributions, idées et travail de la part de la communauté de développement Espresso et nous accueillons volontiers tous les commentaires et suggestions.

Notre but est de continuer à l'améliorer, comme Espresso lui-même s'améliore, pour qu'il puisse continuer à servir d'outil au développement professionnel.

Ce guide peut être lu en séquence, puisqu'il commence par un aperçu sur les outils et les possibilités de Espresso et continue en fournissant les détails de ces outils et comment les mettre rapidement en service dans vos applications.

Ce guide est conçu pour compléter la documentation Espresso, et il est préférable de l'utiliser avec cette documentation, particulièrement la JavaDoc pour Espresso. Le code source lui-même, bien sûr, est la référence ultime sur ce que peut faire Espresso.

PRESENTATION D'EXPRESSO

Expresso est à la fois une architecture et une bibliothèque de composants qui vous aide à construire rapidement et facilement des applications web robustes et riches en fonctionnalités.

2.1 - BIBLIOTHEQUE DE COMPOSANTS

Les domaines individuels d'Expresso peuvent être utilisés de manière indépendante, sans qu'ils ne réclament l'utilisation de Expresso complet. Par exemple, vous pouvez choisir d'utiliser Turbine (une autre architecture ouverte) comme architecture, mais utiliser les objets de base de données d'Expresso comme mécanisme de persistance.

La plupart des composants discutés dans ce guide peuvent être utilisés de cette façon, ce qui fait d'Expresso une précieuse ressource dans tout environnement de développement.

2.2 - ARCHITECTURE

Pour tirer le meilleur parti des possibilités d'Expresso, votre application devrait utiliser Expresso comme architecture. Cela implique d'utiliser le mécanisme d'initialisation de Expresso, sa gestion de configuration et de cache, ses objets de base de données et leurs systèmes de sécurité, son Contrôleur MVC et les objets d'état, avec le mécanisme de présentation de votre choix (JSP, Webmacro ou d'autres).

Utiliser Expresso de cette façon permet encore d'utiliser d'autres outils de développement, d'autres architectures et d'autres bibliothèques de composants. Expresso est conçu pour fournir une extensibilité extraordinaire et peut facilement intégrer beaucoup d'autres outils.

Bien qu'il soit possible d'utiliser les composants d'Expresso sans utiliser tout Expresso, le système rend la construction complète d'applications web plus facile s'il est utilisé comme architecture. Expresso impose des contraintes minimales à une application web. Il ne vous contraint pas d'utiliser un style ou une méthode particulière de navigation ni même une architecture particulière. Les applications web construites avec Expresso comme architecture peuvent utiliser des JSP, des servlets ordinaires, et même des applets ou des applications **stand-alone** comme interface utilisateur.

Dans les chapitres suivants, nous allons surtout parler de l'utilisation d'Expresso en tant que Framework Application, mais les sections individuelles qui décrivent les composants comme les objets base de données, les Contrôleurs, les jobs, etc, s'appliquent tout aussi bien à une utilisation individuelle de ces composants.

Expresso dépend fortement de la disponibilité d'une base de données relationnelle pour ses opérations, et est le plus adapté aux applications pour lesquelles une base de données est une part essentielle.

2.3 - SERVICES

Expresso fournit un certain nombre de services pour l'aide à la construction d'applications web. Voici les principaux domaines couverts par ces services :

⇒ Organisation et configuration d'application

La structure fondamentale d'une application Expresso est contenue dans l'objet schéma de l'application. Cet objet étend la classe "Schema" et agit comme une liste de tous les autres objets qui composent une application donnée. Expresso lui-même utilise une classe Schema (com.jcorporate.expresso.core.ExpressoSchema) pour décrire les classes "communes" à toutes les applications Expresso.

Expresso peut ainsi utiliser l'objet schéma de l'application pour la configurer automatiquement. Le servlet DBCreate lit la liste de tous les objets schéma et peut :

- Créer toutes les tables de base de données demandées par les objets base de données de l'application.
- Fournir les entrées de sécurité par défaut qui permettent l'accès par le groupe Admin à toutes ces nouvelles tables .
- Fournir les entrées de sécurité par défaut pour tous les objets Contrôleur de l'application.
- Fournir les entrées de sécurité par défaut pour tous les objets Job de l'application.

Une fois l'application initialisée, Espresso fournit, pour la gestion de l'application :

- Les entrées Log de l'application peuvent être consultées et le log géré par les servlets de base fournis avec Espresso
- La sécurité de l'application est administrée dans un style Matrix d'emploi facile (par les objets ControllerSecurityMatrix, DBObjSecurityMatrix et JobSecurityMatrix).
- Les jobs de l'application peuvent être mis en file d'attente par le Contrôleur de base QueueJob – ce Contrôleur peut être étendu pour des besoins personnalisés.
- Tous les objets de base de données de l'application peuvent être maintenus (ajout, modification, suppression, recherche) par le servlet de base DBMaint, simplement en lui passant le nom de l'objet de base de données.

2.4 - ACCES AUX BASES DE DONNEES SQL ET AUX BASES DE DONNEES OBJETS

☞ Objets base de données

Les objets base de données utilisent les connexions aux bases de données décrites ci-dessus pour fournir une correspondance aux bases de données relationnelles.

Les objets base de données sont semblables à (et compatibles avec – voir le projet Espresso EJB) **BMP Entity Enterprise JavaBeans**. Ils avaient d'abord pour but de fournir une persistance aux objets métiers, mais qui contiennent fréquemment la logique métier.

Les objets base de données fournissent les méthodes pour ajouter, modifier et supprimer et un certain nombre de méthodes de différentes recherches.

La méthode **searchAndRetrieve** intègre le processus de recherche pour retrouver des enregistrements et un chemin d'accès aux enregistrements trouvés et est fréquemment utilisé pour appliquer un traitement à un ensemble d'enregistrements.

Une extension de l'objet de base DBObject appelée SecuredDBObject utilise une série de tables qui contiennent les informations groupe et utilisateur pour assurer la sécurité base de données au niveau objet.

Les utilisateurs sont rassemblés dans des groupes et ces groupes ne donnent que les permissions nécessaires aux objets base de données.

Les objets base de données qui héritent de SecuredDBObject utilisent automatiquement cette sécurité sans effort supplémentaire de la part du développeur.

☰ Connexions aux bases de données

Expresso facilite la gestion des connexions aux bases de données en rendant le processus de connexion encore plus abstrait que l'API JDBC. Un pool de connexion sophistiqué fournit l'accès à une ou plusieurs bases de données de manière efficace, par les facilités suivantes :

1. Pools multiples :

Une unique classe `DBConnectionPool` fournit l'accès à de nombreux pools de connexion différents, chacun pouvant travailler avec une base de données séparée, et même de éditeurs différents.

Par exemple, vos tables d'Expresso peuvent être stockées sous Oracle, mais vous pouvez avoir un autre pool de connexion avec une base de données DB2/400 sur un serveur différent.

2. Taille maximum.

Une valeur initiale peut spécifier la taille maximum d'un pool de connexion. Quand le pool atteint cette taille, le programme de connexion essaie de nettoyer les connexions périmées pour les réutiliser si possible en cas de nouvelles connexions.

3. Timeout.

Le **pool** de connexion est protégé contre les erreurs d'inadvertance où les connexions ne sont pas relâchées par un mécanisme de **timeout**. Ce **mécanisme retourne une connexion au pool** après un certain délai au cas où le programme client négligerait de relâcher la connexion normalement. Ce **timeout** peut être **override** pour les longues requêtes .

4. Connexions vérifiées

Le **pool** de connexion peut de manière optionnelle vérifier chaque connexion avant de le fournir au programme client à l'aide d'une petite *query* d'exécution sur la connexion, et ainsi tenir compte des situations où la base de données est fermée et empêche le programme client de resté bloqué. Ce mécanisme utilise à la fois une *query* de test et la méthode `IsClosed()`, car la méthode `IsClosed()` peut donner dans certains cas une fausse information. Ce résultat augmente **fortement** la fiabilité de vos applications écrites avec Expresso.

5. Ménage automatique

Le **pool** de connexion nettoie automatiquement les connexions inactives après un certain temps, ce qui réduit la taille du **pool** à un minimum donné pendant les périodes d'inactivité.

6. Configurations spécifiques base de données.

Le **pool** de connexion permet de spécifier des options spéciales base de données en tant que propriétés.

Voir la documentation [fichier Propriétés](#) pour plus de détails.

⇒ MVC Architecture

Espresso comprend un ensemble de composants pour créer plusieurs types d'objets Contrôleur.

Ces objets Contrôleur recouvrent une série d'interaction avec l'utilisateur, de manière semblable à une session TJB (de fait, un Contrôleur peut être une session EJB dans un environnement où les IJB sont supportés). Le Contrôleur peut être utilisé depuis n'importe quelle sorte de client :

- Un servlet
- Un JSP
- Applet
- Une application

Les Contrôleurs sont détaillés dans leur propre chapitre de ce guide.

⇒ Prise en charge et mise en file d'attente de job

Espresso permet de prendre en charge les tâches dont la durée excède le temps d'attente normal d'un utilisateur. Ceci fournit au client un moyen de stocker des entrées dans une file d'attente et au serveur de prendre ces entrées et d'exécuter le processus correspondant.

Cette fonctionnalité est semblable en quelque sorte à l'API JavaSpaces récemment introduite par Sun , mais elle ne réclame pas JINI.

Le *job* est mis en file d'attente par la création d'un objet base de données « JobQueue » et de ses entrées associées « JobQueueParam », qui fournit les paramètres du *job* de la même façon que les paramètres sont fournis à une méthode. Un objet du côté serveur qui hérite de la classe Job est invoqué quand le JobHandler prend un *job* en charge.

Cette classe peut lire les paramètres fournis et au besoin exécuter ses contrôleurs , et normalement signaler par *email* à l'utilisateur qui a initié la tâche quand il a terminé.

☰ Servlets

Expresso fournit le moyen de créer facilement et rapidement des *servlets* utiles et des pages JSP. La création des deux sortes de *servlets* (orientés base de données ou non) est décrite plus loin.

La création des pages JSP qu'utilise Expresso est décrite [ici](#).

Une des plus puissantes possibilités des *servlets* de base de données est que le *servlet* de maintenance de base de données (c-à-d celui qui permet d'ajouter, de modifier et de supprimer dans la table sous-jacente) peut être créé en passant simplement un paramètre au *servlet* DBMaintAuto ; le paramètre est le nom de la base de données, lequel dit au *servlet* DBMaintAuto tout ce qu'il a besoin de connaître pour gérer automatiquement la maintenance de l'objet base de données.

Les classes standard d'Expresso fournissent aussi aux *servlets* le moyen de reporter leurs erreurs et exceptions de manière uniforme et celui de construire des références vers d'autres *servlets* (c-à-d l'action des formes, boutons, etc).

☰ Evénements

Expresso fournit aussi un mécanisme pour définir des « événements » qui se produisent dans l'application WEB.

Un exemple d'événement est la survenue d'une erreur système ; c'est un événement prédéfini dans le système. Cet événement peut être associé à une liste d'utilisateurs qui sont prévenus quand l'événement surgit ; en cas d'erreur système, vous voudrez probablement prévenir votre administrateur système.

Le développeur n'a qu'à spécifier qu'un événement est **suivi** et il est alors automatiquement signalé.

☰ Contrôleurs

Expresso contient un paquet de composants pour créer plusieurs types d'objets contrôleur. Ces objets contrôleurs encapsulent une série d'interactions avec l'utilisateur, de manière similaire à **une session EJB** (de fait, un contrôleur peut être une session EJB dans un environnement qui supporte EJB).

Le contrôleur peut être utilisé depuis toutes les sortes de client : JSP, applet ou application. Pour apprendre à utiliser les objets contrôleurs, voir [ce document](#) .

☰ Utilitaires

Un certain nombre d'autres utilitaires qui ne tombent pas dans les catégories ci-dessus, sont cependant très utiles. En voici quelques uns :

1. Manipulation de fichier : outil pour copier et déplacer des fichiers et faire certaines opérations sur les noms de fichier.
2. Utilitaire de chaînes : quelques outils de manipulations de chaîne plus complexes que ce que permet l'objet String standard de Java.
3. *Logging* : des classes pour maintenir un fichier *log* HTML **color-coded**, avec un niveau de détail ajustable.
4. Processus d'opération système : des classes pour rediriger un appel vers un exécutable externe et pour attendre un temps donné qu'il ait terminé.

Une des références les plus détaillées pour Espresso est la **JavaDoc** que vous pouvez télécharger avec le source de Espresso ; le code source étant bien sûr la référence ultime !

La suite de ce document fournit l'information nécessaire pour commencer à utiliser Espresso.

DEVELOPPER UNE APPLICATION EXPRESSO

Ce guide a pour but de décrire le procédé de développement d'une application Expresso. Bien qu'il y ait beaucoup d'autres façons différentes d'utiliser les composants d'Expresso, les étapes décrites ici sont optimales du point de vue de la conception du système et devraient vous fournir les meilleurs résultats.

Développer une application Expresso **consiste dans** les étapes générales suivantes :

⇒ Planifier l'application

Une application Expresso se constitue normalement d'une série d'objets DB (englobant le modèle et la **persistance** de l'application), d'objets *Controller* (qui fournissent les aspects *Controller* de l'architecture MVC et qui encapsulent toute interaction avec les utilisateurs), les objets Job (qui gèrent toutes les tâches de longue durée et les travaux en tâche en fond) et un unique objet Schéma pour les lier tous ensemble.

Bien sûr, l'application peut avoir beaucoup d'autres objets qui ne tombent pas dans ces catégories, mais celles-ci sont les principales classes d'objets qui concernent Expresso et peuvent être utiles.

⇒ Créer le paquet

Une application Expresso se situe en général dans un paquet unique et des sous-paquets. Par exemple, si vous créez une application de gestion du personnel, vous pouvez appeler votre paquet `com.votresociete.personnel`.

Le paquet de base contient habituellement les paquets suivants :

1. **dbobj** : contient les objets de base de donnée de l'application
2. **controller** : contient les objets *Controller* de l'application
3. **server** : contient les objets serveurs de l'application , y compris les jobs et les programmes **standalone** (c-à-d **en ligne de commande ou basés GUI**) liés à l'application
4. **servlet** : tous les *servlets* de l'application ; de manière typique, il est mieux et plus flexible de relier les objets *Controller* et les *servlets* qu'ils accèdent (*ControllerServlet*, *ControllerActionServlet* et *XMLController*)

☰ Créer un objet Schema pour l'application

L'objet Schema de l'application permet à l'application entière d'être référencée par Expresso. Expresso peut alors générer automatiquement la sécurité de l'application et d'autres tâches administratives à partir du moment où l'objet Schema est enregistré (voir la page «Application» de Expresso pour les détails sur l'enregistrement de l'objet Schema dans Expresso). D'habitude, on crée l'objet Schema dans le paquet « racine » de l'application.

☰ Créer les objets base de données nécessaires

Les objets de base de données sont normalement créés dans le sous-paquet *dbobj* de l'application. Une application Expresso peut utiliser autant d'objets base de données que nécessaire et chacun doit être cité dans l'objet Schema pour que les utilitaires *DBTool* et *DBCreate* puissent automatiquement créer pour vous les tables **voulues** moment où l'application est connectée pour la première fois à une base de données particulière.

Les objets de base de données sont conçus pour être **le chemin persistant** d'Expresso et devraient utiliser pleinement les facilités fournies par Expresso comme le cache, les valeurs autorisées, l'intégrité référentielle et la sécurité native – voir la page dans document sur les objets de base de données pour comprendre comment utiliser ces objets pour en tirer le meilleur parti.

☰ Créer des objets Controller

Les objets *Controller* devraient être utilisés pour encapsuler toute interaction utilisateur exigée par une application Espresso. Une fois encore, l'idée est de profiter de ce que fournit le **framework** et de ne pas réinventer ce qui existe déjà.

Vous devez concevoir votre interface utilisateur d'une façon indépendante de la présentation en pensant en termes d'entrées/sorties abstraites pour profiter des objets *Controller*. Ceci permet d'exploiter au mieux les possibilités d'interface utilisateur GUI-indépendantes à la fois dans Espresso et dans le projet XML Espresso.

La page qui explique les détails des objets *Controller* peut vous aider à créer et à tester facilement vos *Controllers*.

☰ Créer des objets Job

Les objets job sont dédiés aux tâches indépendantes de l'utilisateur, c-à-d les tâches de longue durée qui ne demandent pas d'intervention de l'utilisateur ou les tâches qui demandent beaucoup d'accès aux bases de données et autres sources de données et qui peuvent prendre du temps.

Un objet job typique est conçu pour signaler à l'utilisateur qui l'a initié quand il a terminé. Les méthodes natives des objets job rendent cela facile.

La page qui donne les détails de l'objet job peut vous aider à créer et à **déployer** vos jobs de manière efficace.

OBJETS DE BASE DE DONNEES

4.1 - POURQUOI UTILISER DES OBJETS DE BASE DE DONNEES ?

⇒ Intégrité / consistance de données

En mettant l'accès logique directement dans l'objet de base de données, vous obtenez le même avantage qu'en utilisant des procédures stockées pour accéder à une base de données sans la dépendance de la plate-forme.

L'intégrité référentielle devient indépendante de la base de données et les relations complexes entre les objets de base de données deviennent portables.

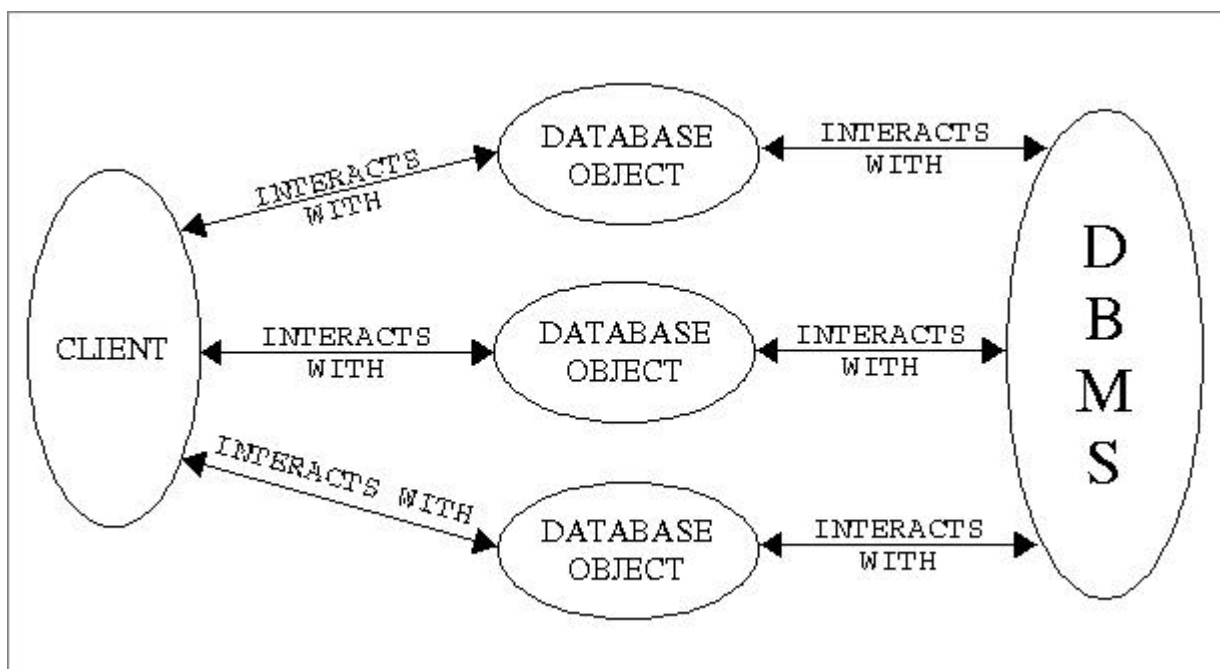
Par exemple, les règles de travail peuvent être intégrées aux objets de base de données et ainsi toutes les applications qui accèdent à ces objets sont assurées de suivre les règles établies.

⇒ Champs virtuels

En ajoutant des champs virtuels (c-à-d des champs calculés) aux objets de base de données et en utilisant des objets "emboîtés", des données stockées en fait dans plusieurs tables peuvent être traitées par l'application comme un seul objet. Par exemple, si un champ virtuel dans un en-tête de facture donne le total de la facture, l'application n'a pas besoin des objets détail pour avoir l'information – c'est un détail d'implémentation qui est caché par l'objet de base de données Facture.

⇒ Entité EJB sans EJB de l'API JAVA

Les objets de base de données fournissent une fonctionnalité similaire à une extension de la spécification d'entité EJB, mais ne réclament pas d'utiliser EJB et la complexité inhérente d'un serveur d'application. Là où sont utilisés les EJB, les objets de base de données peuvent être implémentés facilement comme entités EJB, permettant des augmentations et des diminutions en taille et en complexité.



4.2 - CAPACITES D'OBJET DE BASE DE DONNEES

Les objets de base de données (plus spécialement les objets SecuredDBObject) possèdent un certain nombre de capacités natives utilisables par l'application cliente. Quelques unes sont décrites ci-dessous:

☞ Nombre maximum d'enregistrements

La méthode `setMaxRecords` peut être utilisée pour dir à un objet de base de données que les appels consécutifs pour retrouver plusieurs objets (comme `searchAndRetrieve`) ne doivent retourner qu'un certain nombre d'objets – ce qui est utile pour ne montrer que les n-premiers enregistrements correspondant à une requête , ou fournir d'autres fonctions de gestion de requête pour empêcher l'arrivée d'un trop grand nombre de résultats.

☞ Information statut

L'objet de base de données peut donner au client son propre statut, via l'appel `getStatus()`, pour permettre au client de déterminer si des mises à jour sont nécessaires , si l'enregistrement a été détruit, s'il a besoin d'être enregistré, etc.

☞ Capacité multi-base

Un objet de base de données peut être configuré pour accéder à une autre base de données du même serveur ou une autre base de données sur un serveur différent. C'est très utile pour des scénarios de données **localisées** ou quand les données de contrôle d'Expresso sont dans une base et celles de l'application dans une autre.

☞ Information de la table sous-jacente

L'objet base de données peut fournir le nom de la base de données qui contient les tables de l'application cliente. Ceci permet d'avoir des instructions SQL personnalisées sans sacrifier l'indépendance table et base de données des objets de base de données.

⇒ Change Logging

Il est possible de mettre les objets de base de données en mode de log automatique pour leur faire tracer les modifications de leurs données , ce qui fournit un audit automatique des données critiques sans développement supplémentaire.

⇒ Recherches et ensembles de résultats

Les objets de base de données peuvent être utilisés en mode agrégé où un seul Objet base de données représente une liste d'enregistrements ou d'autres objets de base de données. Ceci permet de manipuler le résultat des recherches.

Il est possible de trier les ensembles de résultat sur n'importe quel champ ou de chercher sur n'importe quel champ, y compris avec des caractères génériques ou des critères d'intervalle. Le nombre d'enregistrements trouvés est obtenu sans accéder à l'ensemble entier.

Il est également possible de ne retourner que les clés des enregistrements pour réduire la taille des données à manipuler.

⇒ Intégrité référentielle *Déclarative*

On peut donner aux objets de base de données une intégrité référentielle entre eux au moyen de simples appels de méthode d'une ligne dans les objets eux-mêmes.

Ces contraintes d'intégrité sont alors vérifiées automatiquement pour toutes les opérations sur ces objets.

Il est aussi possible d'implémenter facilement des suppressions et des mises à jour en cascade basées sur ces contraintes référentielles.

⇒ Ajouter des enregistrements

Ajouter un nouvel enregistrement avec un objet de base de données consiste seulement à remplir les champs de l'objet (qui sont validés de manière appropriée) et à appeler la méthode add() pour enregistrer le nouvel objet.

Les contraintes de clés sont automatiquement observées.

Les mises à jour et les suppressions sont tout aussi faciles.

☰ Concordance de types

Tous les types de données java.sql.Types sont supportés, et peuvent correspondre à tout type de données de n'importe quelle base de données pour une indépendance DBMS totale.

Les champs sont accédés en tant que type de données java approprié par une conversion automatique. Des méthodes spéciales rendent plus facile la prise en compte des champs date/heure

☰ Champs à valeurs multiples

Les champs peuvent être spécifiés comme étant à valeurs multiples, auquel cas les valeurs autorisées de ces champs sont renvoyées par un appel à la méthode *getValues*.

Ces valeurs valides peuvent être cachées pour la performance (via la classe *ConfigManager*) et peuvent être retrouvées à partir d'un autre objet ou calculées, au choix.

Des méthodes spéciales ajoutent très facilement des contrôles de validation à la fois pour les champs à valeurs multiples et pour les champs ordinaires.

La validation est appliquée à tous les accès l'objet de base de données, assurant la validité des données.

☰ Descriptions de champ

L'objet de base de données peut retourner une information étendue sur ses champs, dont une description détaillée (autre que le nom de la base de données).

☰ Tables auto-générées

Un objet de base de données peut créer automatiquement ses tables nécessaires dans la base sous-jacente, ce qui élimine le besoin d'exécuter des scripts SQL pour générer la structure de base de données.

En plus, des outils sont fournis pour la rétro-ingénering d'une base de données existante et générer le code source Java des objets de base de données qui décrivent les tables de la base.

⇒ Lookup Objects

Tous les champs d'un Objet base de données qui est validé par les valeurs d'un autre Objet base de données peuvent avoir leur référence enregistrée.

L'application client peut demander le nom de l'objet référence , peut-être pour fournir une liste des valeurs valides ou pour afficher la relation entre les objets.

⇒ Champs en lecture seule

Les champs peuvent être mis en lecture seule s'ils ne sont manipulés que dans l'objet de base de données.

L'objet de clé séquentielle est un exemple de champ en lecture seule.

⇒ Nombre séquentiels

Un objet de base de données peut générer son propre numéro de série unique. Pour les bases de données qui fournissent cette possibilité de numéro de série séquentiel , elle peut être utilisée, et pour celles qui ne la fournissent pas, la fonction de simple *NextNumber()* est fournie.

⇒ Test Harness

Les objets de base de données ont une méthode native "verify" utilisable à la fois pour tester l'objet et pour contrôler les contraintes établies d'intégrité référentielle ou autre.

Cela rend facile la validation d'une base de données quand de nouvelles règles sont établies ou quand les règles existantes sont modifiées.

4.3 - UTILISATION D'OBJET DE BASE DONNEES

Pour obtenir la plus grande souplesse dans vos application Espresso, il est important d'utiliser les objets de base de données pour accéder aux données physiques et d'empêcher le plus possible l'utilisation d'instructions SQL directes. Ce document va vous aider à faire le meilleur usage des objets base de données et va expliquer les avantages de chaque choix.

La façon la plus directe d'utiliser les objets base de données est d'étendre la classe `com.javacorporate.dboj.DBObject`. Votre classe n'a plus besoin que d'implémenter un petit nombre de méthodes simples pour décrire sa relation avec une table de la base de données. Un raccourci est d'utiliser l'outil DBTool pour générer le code en **réro-Ingeneering** la base de données. Voir la documentation de DBTool pour le détail.

Une fois que vous avez écrit votre Objet base de données, vous pouvez l'utilisez immédiatement dans vos programmes pour accéder à la table.

Un objet base de données doit implémenter au moins les constructeurs pour `DBObject`s et une méthode `setupFields`. Dans la méthode `setupFields`, un certain nombre d'appels de méthode sont utilisés pour établir la relation entre l'Objet base de données et la base de données et un ou des tables spécifiques. Au minimum, `setupFields` doit appeler la méthode `setTargetTable` pour spécifier une table de la base et un appel `addField` pour spécifier un champ de la table.

Par exemple, supposons traiter un objet base de données d'information client. Disons que la table dans la base a un identificateur client unique, un nom et un type de client. Vous pourriez nommer l'Objet base de données "**Client**" et spécifier une méthode `setupFields` comme ceci :

```
1. public void setupFields() throws DBException {
2.   setTargetTable ("Client");
3.   addField("CustomerID", "int", 0, false, "Customer Identifier");
4.   addField("CustomerName", "varchar", 80, false, "Customer Name");
5.   addField("CustomerType", "char", 2, false, "Customer Type");
6.   addKey("CustomerID");
7. }
```

Examinons cette méthode ligne par ligne :

La ligne 1 établit la signature correcte de la méthode pour compiler l'objet.

La ligne 2 spécifie la *table cible* ou la table de la base de données qui sera (en premier lieu) associée à l'objet base de données. Notez qu'aucun nom de base de données n'est mis devant le nom de la table – ceci permet à l'Objet base de données d'être utilisé pour toute base de données appropriée. Le contenu de l'application Espresso spécifiera la base de données connectée au moment de l'exécution.

La ligne 3 spécifie le premier des champs ou colonnes de l'objet et de sa table correspondante. Bien qu'il soit possible que la table ait des colonnes non spécifiées dans l'Objet base de données, cela n'est pas recommandé.

Dans l'appel à *addField*, sont spécifiés les paramètres:

- **Column Name**: C'est le nom par lequel cette colonne est référencée à la fois dans l'objet et dans la base.

Ce nom doit correspondre au nom de la colonne dans la base, mais gardez à l'esprit que l'Objet base de données peut être utilisé pour créer la table automatiquement, ce que nous verrons plus loin. Nous recommandons la convention de nom montrée ici, avec une initiale en majuscule et un majuscule au début de chaque mot suivant; aucun espace ou souligné. A nouveau, pas de préfixe spécifié. Faites attention, quand vous nommez les colonnes, à ne pas utiliser de mots réservés pour aucune base de données, car même si un mot particulier n'est pas réservé dans votre base de données, il est prudent de ne pas restreindre la portabilité de vos applications. Le nom utilisé doit être unique dans l'Objet base de données, mais n'a besoin de l'être pour toutes les tables de la base. En d'autres termes, vous ne pouvez avoir qu'une seule colonne "CustomerID" dans un Objet base de données, mais d'autres objets peuvent aussi utiliser "CustomerID".

- **Data Type** : Le type de donnée Espresso utilisé pour ce champ.

Ce peut on non être le même type de donnée utilisé par votre base de données –les types de donnée Espresso sont "transformés" à l'exécution en types appropriés à la base de données. Vous pouvez même définir de nouveaux types de donnée Espresso pour profiter de capacités particulières d'une base de données (bien que ceci puisse limiter la portabilité et n'est pas recommande pour cette raison).

- **Field Size** : Certains types de donnée réclament une taille de champ ou une longueur maximum qui ne doit pas dépasser 15. D'autres non, comme "int", par exemple. Si une taille de champ est demandée, elle est spécifiée dans ce paramètre – sinon, il est laissé à 0.

Une méthode *addField* supplémentaire permet de spécifier les tailles pour les champs qui ont deux spécifications de taille, comme les champs "float" dont la première taille est la longueur du champ et la seconde est le nombre de décimales.

- **Allow Nulls** : Ce paramètre booléen indique si des caractères nuls peuvent être stockés dans le champ ; faux indique que les nuls ne sont pas acceptés, vrai indique que nul est une valeur permise pour le champ.

- **Field description** : Courte phrase intelligible de description du champ, souvent utilisée comme en-tête de colonne sur les états. Pour une meilleure localisation, cette description devrait être un appel à la méthode *getString* de l'objet schéma de l'application qui peut choisir le nom approprié dans la langue correcte à l'exécution. Voir la documentation sur l'internationalisation.

Lignes 4 et 5

spécifient les champs supplémentaires de la table – bien que l'ordre dans lequel les champs sont ajoutés n'est pas significatif, il affecte la séquence dans laquelle les colonnes sont listées quand la table est créée depuis l'Objet base de données ou quand on utilise les composants Espresso pour afficher les enregistrements de la table, donc un ordre logique devrait être suivi.

La ligne 6

spécifie le champ de clé primaire de l'Objet base de données. Vous devez spécifier au moins un champ de clé primaire, et vous pouvez en spécifier plus d'un en répétant les appels à *addKey* avec chaque nom de champ qui constitue la clé primaire. Les champs de clé primaire ne peuvent pas autoriser les nuls.

Maintenant que vous avez créé l'objet base de données, vous pouvez l'utiliser dans vos programmes. Nous allons examiner les opérations les plus communes de objets base de données.

☰ Retrouver des enregistrements

Pour retrouver un objet base de données correspondant à une ligne particulière dans la base de données, il faut d'abord initialiser l'objet dans le programme :

```
import com.yourcompany.dbobj.Customer;
.
.
.
    Customer oneCustomer = new Customer();
```

Ceci initialise une instance de `Customer`, appelée `oneCustomer`. Maintenant, pour retrouver un client donné, il faut spécifier une valeur pour le champ clé (ou les champs clés s'il y a plus d'un champ pour former la clé).

```
oneCustomer.setField("CustomerID", "1");
```

Cela spécifie la valeur "1" pour le champ `CustomerID`. Notez que la valeur est spécifiée comme une chaîne de caractères – il y a des méthodes `setField` pour les autres types, mais il est toujours possible d'utiliser les chaînes de caractères – la valeur sera convertie dans le type approprié.

Tout ce qui a été fait l'a été en mémoire, il n'y a aucun accès à la base de données tant que nous n'avons pas dit :

```
oneCustomer.retrieve();
```

Ceci accède à la base (ou au cache – voir plus loin) et retrouve l'enregistrement correspondant à la clé donnée. Maintenant, nous pouvons accéder aux autres champs de l'enregistrements avec `getField(fieldName)`;

```
System.out.println(getField("CustomerName"));
```

Imprimera le nom du client n° 1.

Comment faire si nous ne connaissons pas l'ID du client, mais si nous voulons retrouver des clients par d'autres critères?

La méthode `retrieve()` oblige à renseigner le ou les champs de la clé primaire : elle génère une exception dans le cas contraire. (la plupart des méthodes de `DBObject` génèrent `Dbexception` : vous devez inclure le code ci-dessus dans un bloc `catch/try` pour prendre l'exception ou votre méthode générera une `DBException`).

D'autres méthodes de recherche existent pour lesquelles la clé recherchée n'est pas indispensable.

Par exemple :

```
/* Erase any current values in fields */
oneCustomer.clear();
oneCustomer.setField("CustomerName", "Jones");
if (oneCustomer.find()) {
System.out.println("Jones found!");
}
```

La méthode *find()*, en cas de succès, renvoie vrai et renseigne l'Objet base de données avec les champs de la base qui correspondent au premier enregistrement qui correspond. Si la méthode ne trouve aucun enregistrement, elle renvoie "false" et les champs ne sont renseignés.

☰ Retrouver plusieurs enregistrements

Les objets base de données servent aussi à retrouver des ensembles complets d'enregistrements plutôt que un par un. Par exemple, supposons que nous voulions manipuler tous les clients de type "AB" de la base de données. Nous pouvons écrire le code suivant:

```
1. Customer custList = new Customer();
2. Customer oneCustomer = null;
3. custList.setField("CustomerType", "AB");
4. for (Enumeration e = custList.searchAndRetrieve().element();
5. oneCustomer = (Customer) e.nextElement();
6. /* faites ce que vous voulez avec oneCustomer */
7. }
```

Examinons ce code ligne par ligne :

- Ligne 1:** Nous initialisons un nouvel objet client, custList.
Plutôt que d'utiliser cet objet pour travailler avec un enregistrement, nous lui faisons retrouver une liste d'objets clients.
- Ligne 2 :** Nous déclarons une deuxième instance de client pour y mettre chaque client trouvé. Nous n'avons pas besoin d'initialiser cette instance, aussi est-elle mise à nul pour l'instant.
- Ligne 3 :** Nous initialisons le critère de recherche dans l'objet custList en disant que nous cherchons les enregistrements dont le "CustomerType" est égal à "AB".

Ligne 4: Cette ligne est complexe. Nous commençons une boucle "for" en créant une énumération et en initialisant cette énumération avec le résultat de la méthode *searchAndRetrieve()* de *custList*.

Nous pourrions écrire chaque étape de cette ligne séparément comme ceci:

```
Vector v = custList.searchAndRetrieve();
Enumeration e = v.elements();
while (e.hasMoreElements()) {
    oneCustomer = (Customer) e.nextElement();
}
```

Ligne 5: Là, nous retrouvons chaque objet client individuellement, en réutilisant l'objet *oneCustomer* pour chaque enregistrement. La première passe dans la boucle va contenir le premier enregistrement qui correspond au critère, la deuxième le suivant et ainsi de suite. Nous pouvons utiliser l'information de *oneCustomer* pour faire ce que nous voulons à la ligne 6.

De cette façon, il est possible de traiter un aussi grand nombre d'enregistrements que nécessaire tout en conservant un accès à la base de données aussi efficace qu'une simple recherche. Cela minimise les accès à la base pour une plus grande efficacité.

☰ Données ordonnées

S'il est nécessaire de traiter les enregistrements dans un ordre donné, une autre version de *searchAndRetrieve* peut être appelée avec un paramètre qui spécifie les champs de tri. Par exemple :

```
custList.searchAndRetrieve("CustomerName");
```

retrouve les enregistrements dans l'ordre des noms de clients (ascendant).

Pour spécifier un ordre descendant (inversé) , il faut mettre "desc" à la fin de la chaîne, comme ceci:

```
custList.searchAndRetrieve("CustomerName Desc");
```

Vous pouvez aussi trier par plusieurs champs en spécifiant plusieurs noms de champs séparés par le symbole "|", comme ceci:

```
custList.searchAndRetrieve("CustomerName|CustomerID");
```

Ce qui spécifie que les enregistrements sont retrouvés dans l'ordre du nom de client, mais tous les clients qui ont le même nom sont triés par leur numéro ID.

Si aucun critère de tri n'est spécifié , il n'est pas garanti que les enregistrements soient retournés dans un ordre particulier. Souvent, les bases de données renvoient les enregistrements dans l'ordre où ils ont été insérés, ou dans l'ordre de la clé, mais vous ne pouvez pas compter dessus. Si vous avez besoin des enregistrements dans un ordre particulier, demandez-le en spécifiant le paramètre de *searchAndRetrieve*.

☰ Grands ensembles de données

De manière générale, vous devriez spécifier un critère de recherche aussi précis que possible pour ne retrouver que les enregistrements dont vous avez besoin. Vous pouvez mettre des critères sur autant de champs que vous voulez et les combiner pour créer l'ensemble résultat. Par exemple :

```
custList.setField("CustomerName", "Jones");  
custList.setField("CustomerType", "AB");
```

Ce qui signifie que vous voulez les clients dont le nom est Jones et dont le type est "AB".

☰ Utiliser les intervalles et les caractères génériques

On peut aussi utiliser des intervalles et des caractères génériques dans les critères de recherche, la syntaxe exacte dépend de la base de données utilisée. Voir le fichier de propriétés pour savoir comment mettre les caractères génériques appropriés à votre base de données.

Par exemple :

```
custList.setField("CustomerName", "A%");
```

Spécifie une recherche de tous les clients dont le nom commence par "A".

```
custList.setField("CustomerName", "[A-M]%");
```

Spécifie une recherche de tous les clients dont l'initiale est comprise entre "A" et "M".

Egalement,

```
custList.setField("CustomerID", "BETWEEN 1 AND 20");
```

Spécifie une recherche des clients dont le "CustomerId" est compris entre 1 et 20. Faire attention au critère de recherche peut réduire le nombre d'enregistrements traités et accélérer votre application.

Vous pouvez aussi spécifier qu'une recherche ne retourne qu'un nombre maximum d'enregistrements. C'est utile dans le cas où vous n'avez besoin que , disons, des 100 premiers clients correspondant à un certain critère:

```
custList.setMaxRecords(100);
```

signifie que *searchAndRetrieve* ne retrouvera au maximum que 100 enregistrements même si davantage répondent au critère.

Vous pouvez utiliser la méthode **count()** pour savoir combien d'enregistrements répondent à votre recherche sans rechercher les enregistrements eux-mêmes:

```
custList.setField("CustomerName", "A%");
int ct = custList.count();
System.out.println("There are " + ct + "customers with names starting
```

S'il est nécessaire de traiter un très grand nombre d'enregistrements, vous pouvez utiliser une technique de flag, comme dans cet exemple. Supposons que tous les enregistrements ont un champ "Processed" initialisé à "N" :

```
Customer cust.List = new Customer();
Customer oneCust = null;
custList.setField("Processed", "N");
custList.setMaxRecords(100);
boolean moreRecords = true;
while (moreRecords) {
for (Enumeration e = custList.searchAndRetrieve().elements();
e.hasMoreElements();) {
oneCust = (Customer) e.nextElement();
/* Process Customer */
oneCust.setField("Processed", "Y");
oneCust.update();
}
if (custList.count() == 0) {
moreRecords = false;
} /* if */
} /* while */
```

Le code ci-dessus traitera tous les clients par blocs de 100.

☞ Modifier des enregistrements

Comme vous avez pu le voir ci-dessus, modifier des enregistrements est très facile.

Appelez *update()* sur l'objet que vous avez modifié et les changements seront enregistrés dans la base.

Un seul enregistrement est modifié à la fois et il est toujours plus sûr de le retrouver d'abord (voir la section plus loin sur le contrôle de transaction et les opérations *commit* et *rollback*).

⇒ Ajouter des enregistrements

Ajouter des enregistrements est aussi facile que de les modifier : il suffit de renseigner les champs de l'enregistrement (spécialement la clé primaire) et d'appeler `add()`.

Si un enregistrement existe déjà avec la même clé primaire, une exception sera générée.

⇒ Supprimer des enregistrements

Comme ajouter, supprimer oblige à avoir au moins la valeur de la clé primaire. L'appel

```
oneCustomer.delete();
```

supprime l'enregistrement spécifié par l'objet `oneCustomer` (voir la section plus loin sur l'intégrité référentielle).

Supprimer, comme modifier et ajouter, n'affecte qu'un enregistrement à la fois.

⇒ Clauses where personnalisées

Si vous avez besoin de spécifier des relations "ou" ou d'autres conditions spéciales, vous pouvez dire aux objets bases de donnée de mettre une clause "where" personnalisée sur la requête SQL à exécuter. Par exemple :

```
custList.setCustomWhereClause("CustomerType = \"AA\" \ OR CustomerType
```

La clause `where` n'a d'effet que pour la prochaine exécution de la requête; elle est réinitialisée après chaque exécution de la requête pour plus de sûreté.

⇒ Spécifier des bases de données

Jusqu'à présent, tous les exemples que nous avons vus des objets DB assument qu'ils accèdent à la base de données par défaut définie au déploiement d'Expresso. Les objets bases de données peuvent accéder à toutes les bases de données du contexte en utilisant la méthode `setDBName(String)`.

Cette méthode prend en argument le code de la base de données qui est le même que le nom du fichier `.property` pour ce contexte d'Expresso. Par exemple:

```
custList.setDBName("oracle");
```

signifie que l'objet `custList` doit utiliser le contexte "oracle", lequel sans doute a été défini pour connecter une DBMS Oracle.

La plupart des autres objets Espresso permettent de retrouver le contexte actuel de la base de données par l'appel `getDBName()` - Les objets Contrôleur, par exemple, le font. Aussi, un Objet base de données utilisé dans un contrôleur dit simplement:

```
1. custList.setDBName(getDBName());
```

Pour spécifier qu'il doit utiliser le contexte de base de données du Contrôleur- ceci rend l'objet Contrôleur entier portable sur les bases de données.

C'est une bonne pratique de toujours mettre le contexte de la base de données d'un Objet base de données juste après son initialisation.

☰ Utilisation de DBMaint

Le servlet travaille sur tout SecuredDBObject, passé en paramètre et peut exécuter un certain nombre de commandes. Par exemple :

```
1. DMBMaint?dbobj=com.yourcompany.dbobj.Customer&cmd=List
```

Affichera la liste de tous les enregistrements (selon une taille de page spécifiée dans Espresso et discutée plus loin) de la table correspondante à l'Objet base de données Customer. (en supposant que l'utilisateur qui fait la demande a la permission de recherche sur Customer.)

La liste des options de "cmd" (extensible en codifiant de nouveaux objets dans la paquet `com.javacorporate.common.commands`) comprend :

- **Search** : Présente une fiche où l'utilisateur entre un critère de recherche.
- **Add** : Présente une fiche vide où l'utilisateur entre un nouvel enregistrement.
- **Update** : (demande aussi le paramètre clé) Présente un enregistrement à modifier par l'utilisateur.
- **List** : Liste des enregistrements (ou du résultat de requête) et possibilité pour l'utilisateur de choisir un enregistrement à modifier.

Toutes les fiches présentent des icônes permettant à l'utilisateur de changer de mode, ce qui fait de DBMaint une application complète de maintenance de base de données. Les propres pages d'administration d'Espresso sont un bon exemple de l'utilisation de DBMaint.

☰ Contrôle de Transaction

Pour écrire des applications sophistiquées, vous aurez parfois besoin de contrôle de transaction, c'est à dire de la possibilité de réaliser plusieurs opérations de base de données toutes enregistrées en cas de succès et aucune en cas d'échec de l'une des opérations.

Jusqu'à présent, nos exemples d'objets de base de données s'appuient sur la capacité des objets à gérer leur propre connexion à la base. Nous avons autorisé les objets base de données à se connecter au pool de connexion approprié et à relâcher la connexion automatiquement sans spécifier le contexte correct de la base de données.

Le contrôle de transaction nous oblige à spécifier une base de données particulière qui nous permet d'accéder au `commit()` de la connexion.

```
1. DBConnectionPool myPool = null;
2. DBConnection myConnection = null;
3. try {
4. myPool = DBConnectionPool.getInstance(getDBName());
5. myConnection = myPool.getConnection();
6. myConnection.setAutoCommit(false);
7. Customer oneCustomer = new Customer(myConnection);
8. Invoice oneInvoice = new Invoice(myConnection);
9. /* populate the Invoice fields */
10. oneCustomer.setField("Balance", new Balance);
11. oneInvoice.add();
12. oneCustomer.update();
13. myConnection.commit();
14. } catch (DBException de) {
15. if (myConnection != null) myConnection.rollback();
16. throw new DBConnection(de);
17. } finally {
18. if (myPool != null) {
19. myPool.release(myConnection);
20. }
21. }
```

Les lignes 1 et 2 déclarent le pool de connexion et les objets de connexion que nous allons utiliser, lesquels sont déclarés en dehors du bloc try/catch pour être disponibles dans les blocs "catch" et "finally".

La ligne 3 débute le bloc try. Toutes les opération du bloc doivent réussir ou aucune ne sera faite. Pour l'exemple, disons que nous créons une nouvelle facture et une nouvelle balance client quand la facture sera créée. Si la facture ne peut pas être enregistrée correctement, le client ne doit pas être modifié et, vice-versa, si la balance du client ne correspond pas au total des factures de ce client.

Ligne 4 Sur cette ligne, nous demandons une référence à l'objet de pool de connexion approprié de la classe de pool de connexion. Nous lui passons le nom du contexte de la base de données.

- Ligne 5** Nous demandons la connexion au pool et...
- La ligne 6** dit que cette connexion ne fait pas de commit automatique, mais attend l'appel de commit(). Ce qui commence effectivement la transaction.
- Lignes 7 et 8** Installation des objets bases de données Customer et Invoice en passant l'objet de connexion à utiliser. Les objets utiliseront la connexion qui leur est passée plutôt que la leur.
- Lignes 9 et 10** Nous supposons que le code est approprié (peut-être un appel à une méthode) pour renseigner les champs de l'objet Invoice et calculer la nouvelle balance. Disons que la nouvelle balance est stockée dans la variable "new Balance". Cette balance est mise dans l'objet Customer à la ligne 10 (en supposant qu'un champ de nom "Balance" a été défini dans l'objet Customer).
- Lignes 11 et 12** concernent la logique de la transaction. L'objet Customer est modifié et la nouvelle facture est enregistrée dans la base. Si l'une de ces opérations échoue, elle génère une exception est l'exécution se continue à la ligne 15.
- La ligne 13** appelle *commit()* qui confirme les deux opérations.
- Lignes 15 et 16** couvrent la situation où l'une des opérations a échoué. La méthode *rollback()* assure qu'aucune opération partielle ne sera enregistrée dans la base et re-déclenche l'exception pour que la prise en charge d'erreurs de Espresso la traite de manière appropriée.
- Lignes 18 et 19** sont exécutées dans les deux cas de succès ou échec et sont très importantes : sans ces lignes, la connexion ne serait jamais relâchée et le pool de connexion serait vite saturé.

☰ MultiDBObjects

Il est souvent nécessaire de traiter des jointures entre les tables dans une base de données relationnelle – l'objet `MultiDBObject` est fait pour ça. Si un `DBObject` est analogue à une table, un `MultiDBObject` est analogue à une vue (de tables jointes). Beaucoup d'opérations disponibles avec des objets bases de données le sont aussi avec des objets `MultiDB` – dont *searchAnRetrieve()*, *clear()*, *setField* (bien qu'avec des arguments différents), etc.

Cependant, pour renseigner un MultiDBObject, on n'utilise pas *addField*; à la place, il faut ajouter l'Objet base de données entier par la méthode *addDBObject*.

Au contraire de DBObject, MultiDBObject n'est pas une classe abstraite; vous pouvez instancier directement des objets MultiDBObject, au lieu d'avoir à le sous-classer.

Par exemple:

```
1. MultiDBObject myMulti = new MultiDBObject();
2. myMulti.setDBName(getDBName());
3. myMulti.addDBObject(
    "com.jcorporate.expresso.services.dboj.UserDBObject");
4. myMulti.addDBObject(
    "com.jcorporate.expresso.services.dboj.UserGroup", "group");
5. myMulti.addDBObject(
    "com.jcorporate.expresso.services.dboj.GroupMembers", "members");
6. setForeignKey("members", "UserName", "User", "UserName");
7. setForeignKey("members", "GroupName", "group", "GroupName");
8. MultiDBObject oneMulti = null;
9. myMulti.setField("User", "Username", "Fred");
10. System.out.println("User Fred belongs to the following
    groups:");
11. for (Enumeration e = myMulti.searchAndRetrieve().elements();
    e.hasMoreElements()); {
12. oneMulti = (MultiDBObject)e.nextElement();
13. System.out.println(oneMulti.getField("group", "Descrip"));
14. }
```

les lignes ci-dessus sont coupées pour plus de clarté, mais vous n'avez pas besoin de le faire dans une application.

La ligne 1 instancie l'objet MultiDBObject .

Les lignes 3 à 5 spécifient les objets base de données que cet objet "contient" et donnent un nom "court" à ces objets. Par exemple :

`com.jcorporate.expresso.services.dboj.UserDBObject`

est référencé sous le nom "user".

Les lignes 6 et 7 établissent la relation entre les 3 objets en spécifiant un objet clé, un champ et une clé primaire d'un autre objet. Il y a deux relations comme ceci dans notre exemple, le champ Username de l'objet "members" doit correspondre au champ "Username" de l'objet "user" et le champ GroupName de l'objet "Members" doit correspondre au champ GroupName de l'objet groups.

La ligne 8 déclare un nouvel objet MultiDB pour contenir le résultat de la requête, tout comme un Objet base de données.

La ligne 9

positionne le critère de recherche; vous noterez que le nom court de l'objet, le nom du champ et sa valeur doivent être spécifiés; cela fait savoir à l'objet quel champ dans quel objet a cette valeur.

Les lignes 11 à 13

retrouvent le résultat de la requête, comme les objets base de données; sauf que l'appel *getField* prend aussi le nom "court" de l'objet pour lequel trouver la valeur du champ recherché. Vous pouvez aussi utiliser *MultiDBObject* en étendant l'objet prédéfini, en implémentant la méthode *setupFields()*, comme pour les objets bases de données réguliers, mais au lieu d'appeler *addField*, il fait appeler *addDBObject* et *setForeignKey*. Pour l'instant, les objets *MultiDB* sont en lecture seule; c-à-d qu'aucune opération de mise à jour n'est supportée, mais elles le seront bientôt.

☰ Champs virtuels

Les objets base de données ont parfois besoin de présenter des champs qui ne sont stockés dans la table. Ils peuvent être calculés (comme un total de facture) ou recherchés dans d'autres tables (comme un code).

Par exemple, vous pourriez définir un Objet base de données *Invoice* avec un champ pour le nom du client, pour le montrer à la personne qui entre la facture, mais il n'est pas bon de stocker le nom du client dans la facture (normalisation non correcte).

A la place, vous pouvez définir un champ virtuel dans l'Objet base de données et fournir la logique pour rechercher la valeur. Il faut ajouter un appel à la méthode *setupFields* :

```
1. addVirtualField("CustomerName", "varchar", 30, "Customer Name");
```

Dans l'exemple précédent, nous avons spécifié que *Customer* a un champ *CustomerName*. Nous voulons que notre champ virtuel recherche le nom correct automatiquement, comme si le nom était stocké dans la facture.

Pour ça, il faut étendre la méthode *getField(String)*:

```
1. public String getField(String fieldName) throws DBException {
2.     if (fieldName.equals("CustomerName")) {
3.         Customer ourCustomer = new Customer();
4.         ourCustomer.setDBName(getDBName());
5.         ourCustomer.setField("CustomerID", getField("CustomerID"));
6.         ourCustomer.retrieve();
7.         return ourCustomer.getField("CustomerName");
8.     }
9.     return super.getField(fieldName);
10. }
```

Comme vous pouvez le voir, il est important de retourner à tous les autres champs la valeur de la méthode de superclasse.

☰ Champs en lecture seule

Quelques champs peuvent avoir une valeur qui n'est renseignée qu'à la création de l'enregistrement, comme une date de création ou un numéro de série séquentiel. La méthode *setReadOnly* permet de spécifier cette caractéristique. Par exemple :

```
setReadOnly("CreationDate");
```

Si vous avez besoin d'un numéro de série, comme un numéro de facture, vous pouvez utiliser l'objet Expresso NEXTNUM.

Par exemple :

```
public void add() throws DBException {
    NextNumber myNext = new NextNumber();
    myNext.setDBName(getDBName());
    long myJobNum = myNext.getNextTentative("JOB");
    setField("JobNumber", new String().valueOf(myJobNum));
    super.add();
    myNext.getNextConfirm("JOB");
} /* add() */
```

☰ Champs à valeurs multiples

Certains champs peuvent n'avoir qu'un ensemble de valeurs autorisées. Un exemple simple est un champ de type Oui ou Non. Les seules valeurs possibles sont Y ou N. Pour signaler cette restriction, il faut appeler la méthode *setMultiValued* dans *setupFields()*:

```
1. setMultiValued("AccountOpen");
```

Une fois spécifié que le champ est multi-valué, vous devez énumérer les valeurs possibles par l'une de ces deux méthodes :

Par la première méthode, vous étendez la méthode *getValidValues()* pour fournir les valeurs possibles du champ. C'est la méthode la plus appropriée pour les valeurs statiques, comme dans l'exemple ci-dessous :

```
1. public Vector getValidValues(String fieldName) throws DBException
2. if (fieldName.equals("Account Active")) {
3.     Vector myValues = new Vector();
4.     myValues.addElement(new ValidValue("Y", "Yes");
5.     myValues.addElement(new ValidValue("N", "No");
6.     return myValues;
7. }
8. return super.getValidValues(fieldName);
9. }
```

Une autre méthode consiste à spécifier un "objet de recherche" pour le champ. Cette méthode convient pour un champ dont la valeur provient d'un autre Objet base de données, tel un code . Par exemple :

```
1. setMultiValued("CustomerType");  
2. setLookupObject("CustomerType", "com.yourcompany.dbobj.CustomerType");
```

Ces lignes supposent que l'objet CustomerType existe et implémente la méthode `getValues()` qui retourne une liste de valeurs.

Il existe un raccourci pour implémenter des méthodes `getValues()` où les valeurs retournées sont simplement un champ clé et sa description. C'est la méthode `getValuesDefault(String)`.

Les champs Multi-valués sont traités par le programme DBMaint d'un manière spéciale : quand les enregistrements sont listés, la valeur de la description est affichée à la place du code lui-même (c-à-d "Oui" est affiché plutôt que "O") et pour le champ de saisie, un menu déroulant est présenté à l'utilisateur.

☰ Champs secrets

Un champ d'un Objet base de données peut aussi être mis "secret" pour que sa valeur ne soit pas vue par les utilisateurs qui n'ont que la permission "search" – des astérisques sont affichés dans le champ pendant la saisie. Les champs de mot de passe, par exemple, utilisent cette fonctionnalité.

☰ Validation de champ

Il est possible de dire aux objets bases de données quelles sont les valeurs acceptables pour les champs. Plusieurs fonctions interviennent :

- **Null/non null:**

Ce paramètre booléen de la méthode `addField` est une forme simple de validation; s'il est à faux, une valeur non nulle est obligatoire pour le champ correspondant.

- **Contrôle de type de donnée:**

Le type de donnée est aussi une forme de base de validation; seules les valeurs du type approprié seront acceptées, même si le champs est renseigné par un appel à `setField(String, String)`.

- **Valeurs valides:**

Les multi-valués sont aussi validés; seule est permise une valeur de la liste des valeurs valides.

• setField étendu:

Une façon plus spécifique de valider des champs est d'étendre la méthode setField, comme ceci :

```
1. public void setField(String fieldName, String fieldValue) throws
2. if (fieldName.equals("Priority")) {
3. if (!(fieldValue.equals("A") || fieldValue.equals("B") || fieldValue.
4. throw new DBException ("Priority must be A, B or C");
5. }
6. super.setField(fieldName, fieldValue);
7. }
8. }
```

☰ AutoDBObject

L'objet AutoDBObject est le moyen le plus facile d'accéder aux tables de la base de données et est très précieux pour prototyper votre application. AutoDBObject peut renseigner ses champs automatiquement grâce au schéma de sa table. Ce qui permet d'instancier un AutoDBObject et de l'utiliser pour accéder à une table sans aucun code de programmation. Le servlet DBMaint a un paramètre spécial pour permettre d'utiliser un AutoDBObject:

```
DBMaint?dbobj=com.jcorporate.expresso.core.dbobj.AutoDBObject&table=SCHEMALIST&cmd=List
```

Cette commande affiche (et permet d'éditer) la table SCHEMALIST de la base de données courante. Il n'y a besoin d'aucun code pour que l'utilisateur ait accès à l'objet AutoDBoBject (AutoDBObject est un SecuredDBObject).

Note : Donner un accès complet à AutoDBObject rend *toutes* les tables accessibles en lecture/écriture (ou au moins toutes les tables de la base auxquelles à accès un utilisateur donné dans le fichier de propriétés). Cela ne doit être utilisé qu'avec de grandes précautions, surtout en environnement de production.

☰ Cache

Pour augmenter la performance, les objets base de données et les valeurs valides peuvent être "cachées" , c-à-d stockées en mémoire.

Un accès mémoire est de beaucoup plus rapide qu'un accès disque (ou un accès à la base de données) et il peut en résulter une amélioration significative de la performance.

Le cache des objets base de données est activé par une entrée dans la table DBOBjPageLimit en spécifiant une valeur non nulle dans le champ Cache Limit. Ce nombre est le maximum du cache de l'Objet base de données à cacher. Le gestionnaire de cache remplira le cache sur la base du plus fréquemment utilisé, donc la performance s'améliore au fur et à mesure que le cache se remplit.

La meilleure valeur pour la limite du cache de chaque Objet base de données dépend la nature exacte de votre application et de la mémoire disponible pour votre JVM. Les options de la ligne de commande de JVM (-Xmx par exemple) ajustent le total de la mémoire disponible pour le cache.

De plus, les valeurs valides peuvent aussi faire l'objet de cache, et cela améliore aussi la performance. A l'utilisation de la méthode `getDefaultValue`, le cache est automatiquement utilisé.

Les objets cachés sont affectés par les ajouts, modifications et suppressions, pour qu'ils ne se périment pas. Cela fait une autre raison importante n'utiliser que l'accès à la base de données par des objets bases de données ; dans ce cas, les valeurs des enregistrements dans le cache sont toujours correctes. Dans le cas contraire, si vous utilisez une instruction SQL directe comme update, les valeurs dans le cache ne sont plus en accord avec celles de la base de données.

Pour plus de détails sur le cache, voir la documentation du cache.

☰ Objets de base de données dans des bases multiples

Expresso peut définir et maintenir des pools de connexion pour des bases de données multiples et relier des objets bases de données à des sources de données particulières. Expresso doit avoir au moins une connexion à une base de données pour être opérationnel. Cette base primaire est connue comme le contexte de base de données par défaut. La base par défaut contient les informations de configuration Expresso dont Expresso a besoin pour fonctionner. La base par défaut est celle qu'Expresso utilise pour manipuler les objets bases de données, à moins qu'on ne lui dise de "regarder ailleurs" par une directive `otherDB`.

Beaucoup d'applications d'entreprise ont besoin d'accéder à données stockées dans différentes bases. La fonctionnalité `otherDB` permet au développeur de définir un nouveau contexte pour une base particulière et d'associer des objets bases de données avec ce contexte pour manipuler l'objet dans le contexte correct de base de données.

Les applications Expresso peuvent maintenant tourner dans un "contexte partagé", où une base de données contient les tables de contrôle (comme `USERLOGIN`, etc) et une autre base contient les données de l'application (comme un dépôt de données partagé par différentes applications Expresso). Cela permet de créer des "vues" différentes des mêmes données, complètement indépendantes des connexions utilisateurs, des groupes de sécurité, etc. Pour terminer, vous pouvez spécifier plusieurs contextes différents de base de données et définir l'appartenance des objets bases de données à l'un des contextes au niveau du schéma. Alors le système verra toujours la base correcte quand vous utiliserez cet Objet base de données.

Voici comment faire :

- ? Créer un contexte DB, autre que le contexte par défaut. Pour l'exemple, disons que nous créons un contexte DB de nom «hr » pour une base de données de ressources humaines
- ? Créer quelques objets DB pour des tables de la base hr. Par Exemple, nous créons un objet de nom «Employées » pour la table des employés dans la base hr et un objet de nom « Certification » pour la table de certification de la base.
- ? Dans notre schéma, nous définissons les objets DB comme «appartenant » au contexte hr. Ceci se fait en ajoutant l'Objet base de données au schéma de la manière suivante :

- `add (Employee(), "hr") ;`
- `add (Certification(), "hr") ;`

- ? Nous exécutons maintenant DBTool (ou DBCreate). DBTool voit ces directives et créé automatiquement une entrée DBOtherMap pour dire à Espresso de toujours utiliser ces deux objets bases de données dans le contexte DB « hr ». Après l'exécution de DBTool, les deux entrées suivantes apparaissent dans la table Espresso DBOTHERMAP du contexte par défaut :

- `com.mypackagename.Employee | hr | Employee Table`
- `com.mypackagename.Certification | hr | Certification Table`

Comme vous pouvez le voir, la table DBOTHERMAP est utilisée par Espresso pour situer les noms de classes des objets base de données dans un contexte particulier. Cette table peut être manipulée directement, mais exécuter DBTool est en général plus sûr et plus facile.

- ? Les entrées de la table DBOTHERMAP sont lues en mémoire quand Espresso démarre. Et donc, si Espresso tourne déjà quand vous modifiez ces entrées, vous avez besoin de relancer Espresso pour qu'il tienne compte des modifications.
- ? Quand Espresso démarre, vous verrez un message qui dit : «Reading otherdb mappings... », « 2 otherdb mappings found ».
- ? Ces objets seront alors toujours reliés au contexte « hr », plutôt qu'au contexte par défaut.

N.B. : Relier des objets à des contextes OtherDB fonctionnera dans tous les cas si l'objet est autorisé à créer sa propre connexion. Cependant, si vous spécifiez un objet de connexion qui est pointé sur une base de données et que vous créez l'Objet base de données avec cette connexion explicite, l'objet travaillera avec CETTE connexion, sans tenir compte de la table DBOTHERMAP. Quand vous créez un objet avec connexion explicite, vous êtes responsable du fait que la connexion relie à la base de données correcte.

CONTRÔLEUR

Les objets *Controller* fournissent le moyen à une séquence d'interactions d'être disponibles pour virtuellement toutes sortes d'interface utilisateur (c-à-d *Servlet*, *JSP*, *Applet*, application et autres). Un *Controller* est un automate où le flux d'un état à un autre est contrôlé par le *Controller* lui-même.

Un *Controller* peut être sécurisé en utilisant les mécanismes de sécurisation d'Expresso ; un groupe d'utilisateur peut être autorisé ou non à accéder au *Controller* complet ou à certains états seuls du *Controller*. Ceci permet à un unique *Controller* d'être utilisé par un large ensemble d'utilisateurs, car tous ses états ne sont disponibles que pour certains utilisateurs (disons, les administrateurs système).

5.1 - POURQUOI UTILISER CONTROLLER ?

Controller fournit un certain nombre d'avantages :

1. Il encapsule la logique métier séparément de la logique interface utilisateur.

Un *Controller* n'est pas concerné par l'interface utilisateur qui présente les entrées, sorties et actions au client ; ceci sépare les logiques interface utilisateur et métier de manière propre et permet à chacune d'être aussi indépendante que possible, favorisant de bonnes pratiques de **conception**.

2. Accès sécurisé dynamique.

Chaque état d'un *Controller* peut être sécurisé, et les données de sécurité sont facilement maintenues grâce aux capacités natives d'Expresso. Ceci permet aux données de sécurité d'être modifiées de n'importe où, et aux modifications de prendre effet immédiatement. Ceci rend *Controller* idéal pour les situations où les permissions utilisateur seraient modifiées dynamiquement ; par exemple, quand un client termine un *Controller* lui donnant accès à une information en ligne, la fin du *Controller* lui permet d'accéder à de nouveaux états dans d'autres *Controllers*.

3. Architecture MVC.

Les objets *Controller* forment la partie « Contrôleur » de l'architecture MVC de sorte qu'ils sont portables sur tout type d'environnement Java. Ils peuvent gravir tous les échelons entre un simple site basé JSP et un groupe complexe de multi-serveurs qui utilise EJB et des serveurs d'application.

4. Interface utilisateur de base

Un *servlet* spécial, *ControllerServlet*, peut être utilisé pour exécuter *Controller* sans avoir besoin d'un programme ou **design avec un GUI personnalisé de JSP**, autorisant le déploiement très rapide d'un *Controller* de base, amélioré plus tard par un UI personnalisé.

5. Gestion de session

Controller ne préservent normalement aucune information sur leur état d'une invocation à l'autre, ce qui oblige leurs éléments d'entrée à fournir l'information nécessaire pour atteindre l'état suivant. Un *servlet* spécial, *ControllerActionServlet*, peut être utilisé (entre autres commodités) qui préserve l'état entre chaque invocation ; en fait, il préserve le *Controller* courant complet, réduisant l'obligation de recréer l'objet *Controller* lui-même.

6. Test Harnesses

Il existe Test Harnesses pratique pour tester les objets *Controller*, à la fois de la ligne de commande et d'un JSP.

5.2 - CONTROLLER BASICS

Un *Controller* génère trois types d'objets quand il transite d'un état à l'autre.

Ces objets sont :

⇒ Input (entrées)

Un objet **de saisie** est une requête du *Controller* vers le client pour demander de l'information. Quelques attributs supplémentaires de l'objet **d'entrées** peuvent fournir quelques info-bulles formatées, que le client peut ou non utiliser, mais la présentation vue par l'utilisateur est faite par la partie UI de l'application.

Les objets **d'entrées** peuvent aussi fournir «des valeurs valides», ce qui permet à l'UI de présenter à l'utilisateur une liste dans laquelle il choisit. Encore une fois, la nature de la liste est à la charge de l'UI.

⇒ Output (sorties)

Un objet Sortie représente une réponse du *Controller* à présenter à l'utilisateur. Ce peut être aussi simple qu'une chaîne, ou aussi complexe qu'un arbre d'éléments que l'UI peut choisir de représenter par une table, par exemple. Chaque sortie peut contenir plusieurs autres sorties « emboîtées » pour décrire la structure des éléments retournés. Par exemple, une sortie appelée « facture » peut avoir un certain nombre de sorties « ligne élément ».

Les sorties ont aussi des « attributs », qui sont un groupe de paires arbitraires nom/valeur qui aident à décrire encore mieux la sortie à l'application client; par exemple, la sortie « facture » peut avoir un attribut appelé « compte » qui donne le nombre de lignes associées à la facture.

⇒ Actions (actions)

Un objet action représente un choix pour le client de transiter vers un nouvel état; seuls les états appropriés à l'état courant et autorisés par les permissions de l'utilisateur en cours génèrent une élément action.

☰ Blocks (blocs)

Un objet bloc est comme tout autre objet Contrôleur. Son but est d'agir comme un conteneur pour d'autres éléments Contrôleur comme Action, Entrée, Sortie et autres objets blocs.

L'idée sous-jacente à objet bloc est qu'il s'agit d'un groupe logique d'autres objets Contrôleur dans un but de présentation. Pensez à une page HTML qui a de multiples sections (par nécessairement des trames HTML), et où chaque section a ses propres messages, liens et fiches.

Si vous écrivez une JSP, vous devez d'abord avoir une énumération de chacun des objets Entrée, Sortie et Action. C'est alors au développeur JSP de savoir, par le nom, quels éléments Entrée, Sortie et Actions sont à regrouper.

Avec des objets blocs, le développeur peut regrouper chaque ensemble logique d'objets Entrée, Sortie et Action dans un bloc, et chaque section logique de la page HTML s'alloue son propre bloc. Le développeur JSP, de fait, n'aura qu'une simple énumération des blocs et créera le HTML approprié pour chaque objet.

Puisqu'un bloc renvoie chaque élément Contrôleur dans l'ordre où ils ont été mis dans la bloc, si le développeur n'y fait pas attention, les éléments Contrôleur seront placés dans leur ordre de création. Bien sûr, le développeur JSP est toujours libre d'accéder à chaque élément par son nom pour une présentation personnalisée.

Voici un exemple :

Supposons que la page HTML consiste en plusieurs lignes de message, suivies de quelques liens, suivis d'une fiche. Dans un `Contrôleur.someStat()` : Dans le générateur HTML correspondant (qui peut être soit une autre classe utilitaire, un **bean** accédé par JSP, des classes adaptées à WebMacro/Freemarker, ou un JSP), est écrit quelque chose du genre : On peut aussi utiliser l'appel `controller.getBlocks()` pour avoir l'énumération de tous les objets blocs.

Notez que, pour simplifier l'exemple, le mécanisme exact de la génération HTML n'est pas montré ici.

☰ Types de Controllers

Il y a plusieurs types de *Controllers* disponibles, tous étendant la classe de base `com.expresso.corc.controller.Controller.trx`. Ils ont chacun une utilisation spécifique :

- **Controller**

Controller est la classe de base de tous les objets *Controller*, et peut être directement étendue soit vers un *Controller* qui prend en charge ses propres connexions, soit vers un *Controller* qui ne se connecte pas à une base de données.

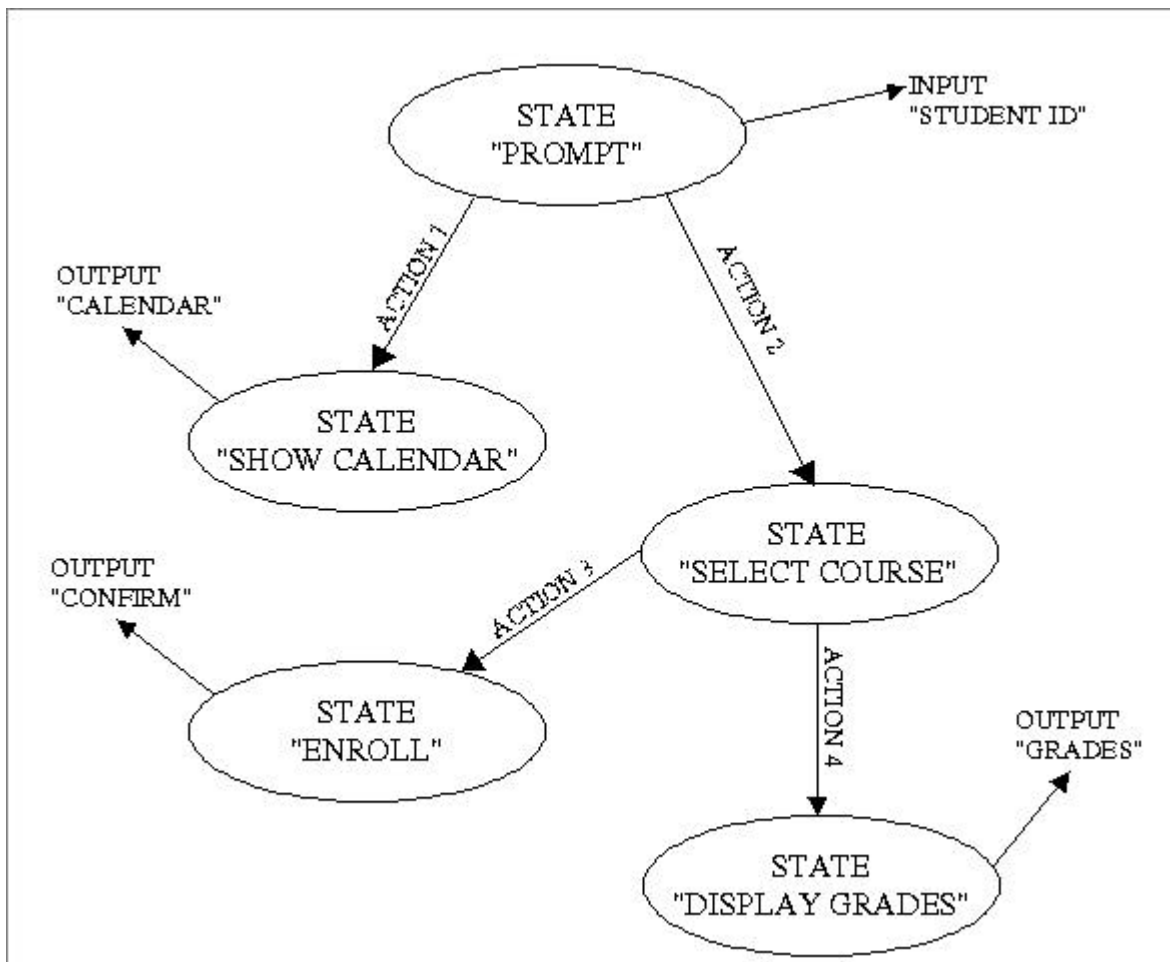
- **DBController**

DBController étend l'objet de base *Controller* et fournit un accès facile à une connexion de base de données à partir du pool de connexion.

- **HTTPController**

Un *HTTPController* étend *DBController* et y ajoute la capacité à accéder aux objets `HttpServletRequest` et `HttpServletResponse` d'un JSP ou d'un objet *servlet*. Ceci permet à un *Controller* d'accéder directement aux paramètres, chaînes de requêtes et autres informations de requête, tout comme aux **set cookies** et aux valeurs de session ; par exemple pour une page de **Login**.

☰ L'activité Controller expliquée



Dans le diagramme ci-dessus, vous voyez une représentation simple d'un *Controller* avec 5 états. Suivons l'exécution de ce *Controller* pour voir comment les interactions travaillent :

1. Le *Controller* commence (disons qu'il est appelé par une page JSP pour les besoins de la discussion) à l'état « prompt » en haut. Entrer dans cet état entraîne le *Controller* à générer un élément de saisie, de nom « Student Id » et deux éléments d'action. Les éléments d'action indiquent que l'utilisateur peut choisir à ce point de sélectionner un cours (Action 2) ou de voir le calendrier (Action 1) (Nous supposons que tout utilisateur connecté a en fait la permission d'aller vers tous les états). Le contrôle retourne alors à l'interface utilisateur.
2. Disons que l'utilisateur entre un « Student Id » approprié et choisit l'action 2, « select course ». Le *Controller* transite de l'état « prompt » à l'état « select course ». L'état « select course » a un paramètre obligatoire, le « Student Id » que le précédent état demandait. Si ce paramètre est absent, la transition vers le nouvel état échoue (et l'utilisateur devrait en être informé de la manière que l'UI a choisi). Dans notre cas, le « Student Id » est présent, et donc le *Controller* entre dans l'état « select course ».

3. Cet état produit un autre élément d'entrée ; cette fois, il demande à l'utilisateur de choisir dans une liste de cours. L'élément de saisie peut être alimenté par une liste de valeurs valides (dans ce cas, les cours disponibles pour cet étudiant, par exemple), et peut être représenté par l'UI sous la forme d'un menu déroulant . l'état « select course » génère aussi deux éléments action , Action 3 et Action 4. Si l'utilisateur ,n'a pas l'autorisation d'accéder à l'état « enroll », par exemple, alors Action 3 ne sera pas généré.

4. Mettons que l'utilisateur choisisse un cours dans la liste et que ce soit Action 4 ; pour transiter à l'état «display grades ». Dans ce cas, l'état «display grades » réclame deux paramètres : le « Student Id » du premier état & le cours choisi dans le second état. Si les deux sont présents, la transition s'effectue et l'état « display grades » génère un élément de sortie (probablement un groupe de sortie utilisant des sous-sorties) qui présente à l'utilisateur l'information demandée.

Cet exemple est bien sûr simpliste et un *Controller* réel aurait beaucoup d'autres connexions entre les états, et probablement un comportement plus complexe dans chaque état, mais il sert à illustrer l'utilisation du modèle *Controller*.

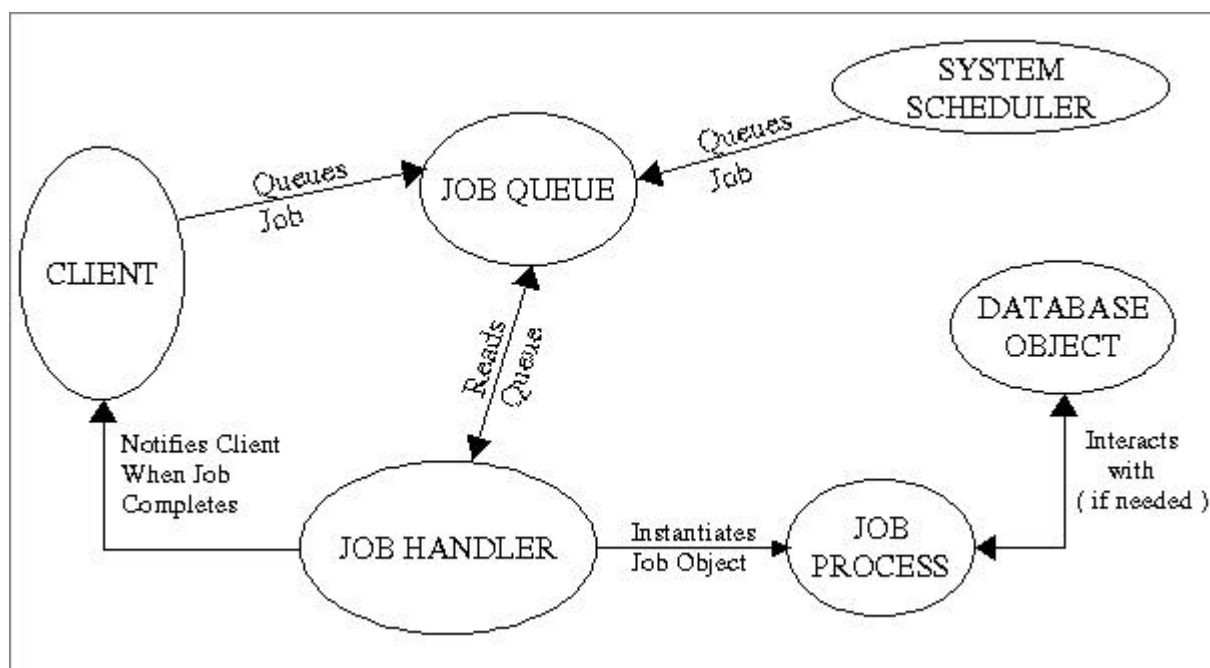
Il est important de réaliser que ce *Controller* peut être utilisé de différentes manières ; par exemple, à partir d'une page JSP qui montre une liste d'étudiants inscrits dans un cours particulier, un utilisateur autorisé pourrait choisir un lien qui dit « voir le classement ». Ce lien peut invoquer notre *Controller* exemple, mais pas à l'état initial : il doit aller directement à l'état « show grades », supposer que les permissions sont correctes et que le cours et le « Student Id » sont renseignés (puisque'ils sont des paramètres obligatoires pour entrer dans cet état) ; notre simple *Controller* répondrait avec la même résultat que celui fourni quand il est invoqué de la façon que nous avons décrite ci-dessus.

Cette sorte d'interaction avec d'autres *Controllers* augmente considérablement l'utilité du modèle *Controller* et en favorise la réutilisation sans sacrifier la portabilité ou l'intégrité des données.

Bien sûr, un objet *Controller* interagit de manière typique avec un ou plusieurs objets base de données (ou d'autres *Controllers*) pour faire son travail ; mais ces interactions sont complètement intégrées dans les états eux-mêmes et le client n'a pas besoin de s'en inquiéter.

LES COMPOSANTS JOB

Expresso facilite la prise en charge de tâche dont la durée excède un délai d'attente raisonnable pour l'utilisateur. Il fournit au client le moyen de stocker une entrée dans la file des jobs et au serveur celui de lire ces entrées et d'exécuter les tâches correspondantes.



6.1 - ECRIRE DES OBJETS JOB

Un objet job **étend** la classe `com.jcorporate.expresso.core.job.Job` et devrait, dans son constructeur, appeler la méthode `addParameter` pour « déclarer » de quels paramètres `JobQueue` il a besoin. Le `JobHandler` peut alors vérifier l'existence de ces paramètres dans la file d'attente des jobs et déclencher le processus d'erreur s'ils ne sont pas spécifiés.

Par exemple, l'objet job `com.jcorporate.expresso.ext.job.SendNotice` déclare :

```
addParameter ("Subject", "Message Subject") ;
addParameter ("FileNumber", "File Number") ;
addParameter ("NoticeText", "Notice Text") ;
```

L'objet `SendNotice` spécifie qu'il réclame les paramètres `Subject`, `fileNumber` et `NoticeText` quand il est mis en file d'attente. L'utilisation de `addParameter` signifie aussi que le *Controller* générique « `QueueJob` » peut être utilisé pour mettre ce job dans la file d'attente sans avoir à écrire une **façade** spéciale pour lui.

Le job devrait aussi implémenter une méthode `getTitle()` afin que les fonctions standards de sécurité puissent **Lister** son nom correctement. C'est aussi simple que :

```
Public String getTitle() {
    return new String("Envoi de Notices");
}
```

La dernière méthode qu'un job devrait implémenter est la méthode `run()`, tout comme une tâche. La méthode `run()` est appelée par `JobHandler` quand le job a été instancié et initialisé, pour exécuter celui-ci. Un job peut interagir avec une base de données, envoyer des email et faire tout ce dont a job a besoin. La méthode `run()` utilise habituellement la méthode `getJobQueueEntry()` de la superclasse pour accéder à ses paramètres et savoir ainsi ce qu'il a à faire, et envoie, par exemple, un email pour signaler qu'il a terminé. Le job `SendNotice` est un bon exemple de job simple qui peut être recopié et modifié.

6.2 - SECURISER LES OBJETS JOB

Les objets job ont des données de sécurité dans la base de données , via l'objet de base de données JobSecurity. Ces données peuvent être utilisées pour spécifier si un utilisateur est autorisé à exécuter un job spécifique ou pour spécifier les « fonctions » du job autorisées pour un utilisateur donné.

La signification exacte de « fonction » est particulière à chaque job ; par exemple, si un job peut soit calculer une hypothèque (fonction « hypothèque ») ou un prêt personnel, un utilisateur donné peut n'avoir que la permission pour la fonction « hypothèque » ; c'est au job lui-même d'avoir l'action appropriée si la mauvaise fonction est demandée.

De cette façon, la sécurité du job est moins sophistiquée que la sécurité fournie par les *Controller*. Bien sûr, il n'y a aucune pour qu'un job n'appelle pas au besoin un *Controller*.

6.3 - METTRE LES OBJETS JOB EN FILE D'ATTENTE

Le job est mis en file d'attente par la création d'un objet de base de données « JobQueue » et ses entrées associées « JobQueueParam » qui fournit les paramètres au job de la même façon que les paramètres sont passés à une méthode.

Un objet du côté serveur qui hérite de la classe job est invoqué quand le processus JobHandler commence à prendre le job en charge. Cette classe peut alors lire les paramètres qui lui sont donnés et exécuter son *Controller* au besoin, et d'habitude, signaler ,via email, à l'utilisateur qui a initié la tâche quand il a terminé.

Il peut , bien sûr, y avoir plusieurs processus JobHandler qui travaillent dans la file d'attente des jobs au même moment et ils synchronisent leurs requêtes pour que seul un processus serveur prenne un job donné. Ceci permet **fail-over and scalability** , car les performances peuvent être augmentées simplement en ajoutant de nouveaux objets JobHandler pour prendre les requêtes en charge plus rapidement.

DEVELOPPER DES JSP

Le développement JSP en utilisant Expresso s'appuie fortement sur l'architecture MVC, dont les composants sont les suivants :

1. Modèle

L'objet *Controller* englobe la partie « modèle » dans Expresso. Il prend en charge toute la partie logique de JSP. Dans un environnement EJB, le *controller* sert à encapsuler la session EJB qui réalise la partie logique courante. (une autre possibilité est que l'objet *controller* peut lui-même être un EJB).

2. Vue

La partie « vue » de Expresso est la page JSP elle-même. Elle ne fournit que le formatage de l'affichage pour l'utilisateur ou celui de la fiche de saisie de l'utilisateur.

3. Controller

Dans Expresso, la partie *Controller* de la logique est largement supportée par le *servlet* *ControllerActionServlet*; il est chargé de passer l'information de la page JSP, par l'intermédiaire du *Controller* de la page, en envoyant automatiquement la fiche des champs de données et les paramètres des requêtes.

DEVELOPPER DES SERVLETS

Cette section concerne les développeurs qui veulent construire leurs propres *servlets* en utilisant les composants standards fournis par Expresso. Il est facile de construire des *servlets* qui héritent de fonctionnalités avancées sans écrire beaucoup de code, comme nous allons le voir.

Il y a fondamentalement deux types de *servlets* qui peuvent être construits à partir des blocs d'Expresso :

1. Servlets de base de données

Les *servlets* de base de données connectent ,via l'Objet base de données `ConnectionPool`, à une base de données relationnelle et peuvent accéder aux objets de base de données (`DBObjects`).

Ces *servlets* héritent de la classe **DBServlet** .

Un exemple d'un tel *servlet* est le *servlet* `RunSQL`, lequel permet de passer des *queries* à une base de données.

2. Servlets non orientés base de données

Les *servlets* non orientés base de données sont ceux qui ne réclament pas une connexion à une base de données, comme le *servlet* `Search` fourni avec Expresso.

Ces *servlets* bénéficient encore d'un certain nombre de services profitables en héritant de l'objet **StdServlet**.

8.1 - SERVLETS NON ORIENTES BASE DE DONNEES

Les *servlets* non orientés base de données bénéficient encore des méthodes de l'objet `StdServlet` quand ils en héritent.

L'objet `StdServlet` fournit les fonctionnalités suivantes accessibles par ses **descendants** :

1. Extraction de paramètre

`StdServlet` lit automatiquement ses paramètres à l'invocation du *servlet* sur la page HTML dans un vecteur prêt alors à être lu par le **descendant**.

2. self-referencing

Les méthodes `getServletPrefix` servent à construire des liens pour appeler le même *servlet* avec des paramètres différents ou un autre *servlet* sur le même système.

3. Gestion d'erreurs

La fonctionnalité la plus significative que fournit `StdServlet` est peut-être la gestion aisée des conditions d'erreur et des exceptions, à travers la fonction `showError`. Il est possible de simplement inclure les méthodes `doGet` et/ou `doPost` dans un `try/catch` ; voir le *servlet* `Search` en exemple.

8.2 - SERVLETS DE BASE DE DONNEES

Les *servlets* de base de données héritent de l'Objet base de donnéesServlet et fournissent quelques services en plus :

1. Connexion à la base de données

Les *servlets* de base de données se connectent à la base de données via le **pool** de connexion. Le *servlet* est responsable de la fermeture de la connexion obtenue dès que possible.

2. Sécurité de l'application

La méthode *checkAppSecurity* permet au *servlet* de déterminer si l'utilisateur connecté a ou non la permission de l'utiliser.

3. Login

Par la méthode *requireLogin*, l'ancêtre de DBServlet peut s'assurer que l'utilisateur connecté est un utilisateur autorisé à accéder à ses fonctions.

4. Valeurs de configuration/setup

Grâce au **pool** de connexion de base de données, l'ancêtre DBServlet a accès aux valeurs de configuration stockées dans la base de données. Ces valeurs peuvent être utilisées, par exemple, pour configurer l'application pour des préférences et des chemins spécifiques à un site.

8.3 - SERVLETS DE MAINTENANCE DE BASE DE DONNEES

Une instance spéciale de *servlet* de base de données qui est aussi encore plus facile à construire en utilisant Espresso est un *servlet* de maintenance de base de données.

Ce genre de *servlet* permet les opérations de maintenance de base sur quelques objets de base de données (souvent une table) dans une base de données relationnelle. Par exemple, une fiche personnalisée de maintenance serait un exemple de *servlet* de base de données de maintenance.

Il y a beaucoup d'exemples de tels *servlets* dans Espresso ; c'est de loin la sorte la plus commune de *servlets* dans le **framework**.

La plupart des *servlets* de base de données de maintenance se contentent d'utiliser le *servlet* de base **DBMaint** pour fournir leurs services. DBMaint utilise l'information stockée dans un Objet base de données pour ajouter, modifier, lister, rechercher et supprimer de façon standard , sans avoir à réécrire le code pour chaque *servlet*.

En utilisant l'extension de DBObject appelée SecuredDBObject, ces fonctions peuvent aussi être sécurisées pour n'autoriser à chaque utilisateur que l'accès approprié à chacune des tables de la base de données.

Le *servlet* DBMaint est conçu pour s'étendre facilement. Chacune des commandes de DBMaint est en elle-même un objet extensible. Ces objets sont stockés dans le paquet `com.corporate.expresso.core.servlet.commands` et peuvent être étendus pour des fonctionnalités personnalisées.

☰ Utilisation de DBMaint

Pour utiliser DBMaint pour fournir immédiatement les possibilités de maintenance de base de données, vous devez d'abord définir un objet de base de données sécurisée (SecuredDBObject) . En pratique, c'est très simple :

1. Décidez dans quel *paquet* sera votre nouvel objet. Il ne devrait pas être dans les répertoires utilisés par Espresso . Nous recommandons un nom de paquet de la forme : « com.corporate.expresso.ext.application.dboj » , où « application » est celle que vous créez ; par exemple, s'il s'agit d'une application de ressources humaines, vous pourriez choisir « hr » , et ainsi, le nom du paquet serait « com.corporate.expresso.ext.hr.dboj » . Le nom de l'objet devrait refléter son but initial : si vous créez un objet pour maintenir une base de données des employés, son nom serait « Employes » .
2. Copiez une définition existante SecuredDBObject en dehors de Espresso pour l'utiliser en tant que **template** ; sauvegardez **some typing** ! Nous recommandons l'objet com.corporate.common.expresso.core. En tant qu'exemple simple.
3. Modifiez l'objet copié en remplaçant toutes les instances de nom « User » par le nom « Employes » , modifiez la définition du paquet, et bien sûr la table de destination et les définitions de champs pour correspondre à ceux que vous voulez avoir dans votre base de données. Consultez la documentation JavaDoc sur l'objet SecuredDBObject pour plus de détails.
4. A ce point, vous pouvez décider de créer manuellement la table qui contient l'objet Employes ; il y a moyen de le faire faire à Espresso, décrit dans la section sur les objets.
5. Compilez votre nouvel objet pour rendre disponible son **fichier classe**.

Nous pouvons maintenant créer un lien avec le nouvel objet base de données utilisant *DBMaint servlet*:

- Sélectionner la page web à invoquer à partir de la fonction de maintenance : supposons qu'il se nomme hr.html
- Sur cette page, créer un lien pour obtenir une liste (LIST), un ajout (ADD), une recherche (SEARCH) pour ce nouvel objet.
Vous pourrez avoir l'utilité des images et format utilisés dans Espresso. Celles-ci peuvent être copiées des pages comme server.html dans /components/expresso.

- Chaque lien doit avoir la forme qui suit :

```
"/servlet/DBMaint ?next=hr.html&dbobj=com.javacorporate.ext.hr.dboj.Employee&cmd=Add"
```

- Le dernier paramètre (cmd) peut avoir 3 valeurs possibles :
 - ADD affiche une page pleine de saisie de nouvelles valeurs pour la table dans la base de donnée
 - LIST affiche tous les enregistrements contenus dans la table
 - SEARCH affiche une page de champs pour saisir les critères de recherche des enregistrements à sélectionner.

L'option LIST doit être utilisée avec précaution. Si le nombre d'enregistrement doit être très important, il existe des facilités pour limiter le nombre d'enregistrement et de prévoir une pagination au cas où. Il existe aussi un argument UPDATE, voir JAVADOC pour les détails de l'Objet DBMaint.

- Créer une entrée objet base de donnée sécurité pour le nouvel objet : voir fichier d'aide pour l' « objet sécurité base de donnée », pour plus d'information et comment le mettre en œuvre.

☰ DBMaint : un exemple pas à pas

Voici un exemple de construction d'une application simple qui permet la maintenance sécurisée d'un **client**. Cet exemple suit le processus décrit ci-dessus, est complété par le code source et est plus détaillé.

Nous assumerons que notre fiche de maintenance client n'est qu'un début pour une application de suivi des ventes plus sophistiquée.

1. Définir l'objet base de données

Supposons pour l'exemple que nous avons installé Expresso dans `/usr/java/lib/com/corporate`. Notre CLASSPATH devrait alors contenir `/usr/java/lib`. (Bien sûr, la CLASSPATH de votre moteur de *servlet* doit aussi contenir ce répertoire ; consultez la section d'installation).

- 1) Nous créons d'abord le répertoire `/usr/java/lib/com/corporate/expresso/ext`, dans lequel nous allons définir les « extensions » du **framework** Expresso **javacorporate**. Le nom du paquet pourrait aussi être en dehors du répertoire `corporate` si on le voulait.
- 2) Dans le répertoire «ext», nous créons un répertoire pour notre paquet, de nom «ventes» (c-à-d que nous créons le répertoire `/usr/java/lib/com/corporate/expresso/ext/ventes`). Dedans nous créons un autre répertoire, de nom «dbobj» pour contenir nos objets de base de données.

Cette longue hiérarchie de répertoires est caractéristique des applications Java et aide à conserver les paquets bien organisés ; encore une fois, nous n'utilisons ces noms particuliers que pour l'exemple, si vous voulez une organisation différente, c'est tout aussi faisable.

- 3) Dans le répertoire `ext/dbobj`, nous créons l'objet de base de données, de nom «client.java». Le code qui suit définit cet objet :

Les commentaires dans le source décrivent le but de chaque section.

```
/*
 * Customer.java
 */

package com.javacorporate.ext.sales.dbobj;

import com.jcorporate.expresso.core.dbobj.*;
import com.jcorporate.expresso.core.db.DBConnection;
import com.jcorporate.expresso.core.db.DBException;

/**
 * A Customer object stores information about customers for our demo
 * sales application.
 */
public class Customer extends SecuredDBObject {

    private String thisClass = new String(this.getClass().getName() + ".");

    /**
     * There are three possible constructors for SecuredDBObjects
     * Usually there is no need to extend them for objects we implement
     */
    public User() throws DBException {
        super();
    } /* User() */

    /**
     * This constructor is used when a connection to the database
     * is supplied by the calling object
     */
    public User(DBConnection theConnection) throws DBException {
        super(theConnection);
    } /* User(DBConnection) */

    /**
     * This constructor is called by the DBMaintServlet
     * and supplies the name of the user trying to connect to the
     * database object.
     */
    public User(DBConnection theConnection, String theUser) throws DBException {
        super(theConnection, theUser);
    } /* User(DBConnection, String) */

    /**
     * The setupFields method does the real work of establishing the
     * definition of the DB object.
     */
    public void setupFields() throws DBException {

        /* Establish the primary database table for this object. */
        /* Note that there may be more than one, but this is the default */
        /* table */
        setTargetTable("CUSTOMER");

        /* Set a description for this object. This is a human-readable */
        /* string that appears at the top of forms when used with DBMaint */
        setDescription("Users");
    }
}
```

```

/* Define each of the fields in the table */
/* Note that the field types used here are "internal", and can be */
/* mapped to other types for the underlying relational database */
addField("CustomerNumber","int", 0, false, "Customer ID Number");
addField("CustomerName","varchar", 80, false, "Customer Name");
addField("Phone", "varchar", 15, true, "Phone Number");
addField("EMail", "varchar", 80, false, "Customer EMail");

/* ... you could of course add many more fields for a production */
/* object */

/* addKey is called for each field in the primary key */
addKey("CustomerNumber");

} /* setupFields() */

/**
 * A utility method used when generating lists of objects
 */
public DBObject getThisDBObject() throws DBException {
    return (DBObject) new Customer(this.getConnection());
} /* getThisDBObject() */

} /* Customer */

```

- Compilez l'objet client.java et corrigez les erreurs.

2. Créer un objet Schéma pour notre nouvelle application

L'objet Schéma identifie l'application entière et ses objets DB auprès du framework Expresso. Les facilités d'Expresso de création de tables et de valeurs de configuration sont alors disponibles, à votre grand avantage, et vous évitent du travail lors du démarrage de l'application.

- Dans le même répertoire « dbobj », créer un fichier java « ventes.java » comme ceci :

```
/*
 * Sales.java
 */

package com.javacorporate.ext.sales.dbobj;

import com.jcorporate.expresso.core.dbobj.*;
import com.jcorporate.expresso.core.db.DBConnection;
import com.jcorporate.expresso.core.db.DBException;

/**
 * Schema object for the Sales demo application
 */
public class Sales extends Schema {

    /**
     * Default Constructor
     */
    public Sales() throws DBException {
        super();
    } /* Sales() */

    /**
     * setConnection does the real work of configuring a schema object
     */
    public void setConnection(DBConnection dbconn) throws DBException {
        super.setConnection(dbconn);

        /* add is called for each object to be a "member" of this schema */
        add(new Application(dbconn));
    } /* setConnection(DBConnection) */

    /* Other methods are optional in the Schema object - see the Javadoc */
    /* for details */

} /* Sales */
```

Ce simple schéma sera étendu que vous ajouterez de nouveaux objets à l'application.

- Compilez l'objet ventes et corrigez les erreurs.

3. Créer les liens dans HTML

Maintenant, il faut créer un lien vers le nouvel objet de base de données en utilisant le *servlet* DBMaint :

- Créer la page WEB qui appellera la fonction de maintenance : soit « ventes.html ».

La présentation de la page n'a pas d'incidence sur le fonctionnement des *servlets* eux-mêmes. Concentrons-nous sur les liens qui appellent notre nouvel objet.

- Sur cette page , créer les liens suivants :

Ajouter un client

1.

```
<a href= »/servlet/DBMaint?next=sales.html
&dbobj=com.javacorporate.ext.sales.dbobj.Customer
&cmd=Add">Add Customer</a>
```

Liste de clients

1.

```
<a href= »/servlet/DBMaint?next=sales.html
&dbobj=com.javacorporate.ext.sales.dbobj.Customer
&cmd=List">List Customer</a>
```

Recherche de clients

1.

```
<a href= »/servlet/DBMaint?next=sales.html
&dbobj=com.javacorporate.ext.sales.dbobj.Customer&cmd=Search
">Search for Customers</a>
```

4. Enregistrer le nouveau schéma

Maintenant, il faut informer Espresso de l'existence de notre nouveau schéma, pour que la fonction d'initialisation accède correctement à cet objet :

1. Dans votre **navigateur** WEB, allez aux pages Espresso que vous avez téléchargées depuis le site Javacorporate.
2. Cliquez sur le lien « Setup » dans la colonne à main gauche.
3. Sur la ligne titrée « Application Schema Objects » , cliquez sur le signe plus (+) pour ajouter une nouvelle entrée.
4. Dans le champ « Schema Class File », entrez « com.corporate.expresso.ext.dbobj.ventes », le nom de l'objet schéma que vous venez de créer.
5. Dans le champ « Schema Description », entrez « Application de ventes »

5. Exécuter « Initialize »

La fonction Initialize créera la table pour votre nouvel objet et les entrées de sécurité de base pour que le groupe Admin (et l'utilisateur Admin) ait accès aux nouveaux objets.

1. Cliquez sur le lien « Setup » dans la colonne de gauche des pages Espresso.
2. Choisissez le *servlet* « Initialize ». Laissez tous les cases cochées et cliquez sur le bouton « Run ».
3. Au bout d'un moment, vous verrez la confirmation que les tables ont été initialisées et que les entrées de sécurité ont été créées.

6. Se connecter en tant qu'Admin

Allez à la page Log In/out et utilisez Login pour vous connecter en tant qu'Admin. Puisque votre nouvel objet n'est encore accessible que par Admin, c'est pourquoi vous devez être Admin pour le tester. Plus tard bien sûr, vous pourrez ajouter des sécurités pour d'autres groupes et utilisateurs.

Maintenant testez votre nouvelle fonction de maintenance de base de données ; allez à la page `ventes.html` et essayez les fonctions Ajout, Liste et Recherche.

Ceci est bien sûr un exemple très simple, mais qui peut être la base d'une application plus complexe.

UTILISER L'UNITE DE TEST DE EXPRESSO

Expresso possède le noyau d'une unité extensible de test de l'*architecture*. Ces classes sont dans le paquet `com.corporate.expresso.core.test`

Pour pouvoir utiliser l'unité de test, faites ce qui suit :

- Assurez-vous que votre classe a un constructeur public par défaut
- Implémentez l'interface `com.javacorporate.expresso.services.test`. Voir la description dans JavaDoc de cette interface pour l'explication des divers paramètres.

Exemple :

```
package expack

import com.jcorporate.expresso.services.test.*;

class Example implements Testable {
    public Example() {}

    public boolean unitTest(String argv[], boolean verbose, java.io.PrintStream out) throws Exception
    {
        if (Test.performTests == true) {
            boolean allPassed = true;
            if ("a".equals("a")) {
                if (verbose == true) out.println("Test Passed ");
            } else {
                allPassed = false;
                if (verbose == true) out.println("Test Failed" );
            }
            return allPassed;
        }
    }
}
```


Exécuter le programme principal de test Espresso avec les paramètres suivants :

`configDir=<chemin d'accès complet à votre fichier de configuration Espresso> class=<nom complet de la classe que vous voulez tester>`

Exemple :

```
Java com.jcorporate.expresso.services.test.ExpressoTest class=expack.Example  
configDir=c:\Expresso\Config
```

☰ Directions futures

L'**architecture** aura aussi un fichier de configuration XML, donc si vous ne passez pas l'argument de classe, le programme de test testera automatiquement toutes les classes du fichier XML avec les paramètres spécifiés dans le fichier test. Ceci donnera un contrôle complet sur **regression testing** qui pourra tester une architecture importante.

Il est également nécessaire de concevoir et d'inclure la possibilité de tester les *servlets*. Pour l'instant rien n'est fait.

DEPLOIEMENT DES COMPOSANTS DE L'APPLICATION EXPRESSO

CONTROLLER LES OPERATIONS DES APPLICATIONS EXPRESSO

Maintenant que votre application est développée et déployée , quels sont les outils qui vous permettent de contrôler et de maintenir votre application ? Vous voulez , par exemple, vous assurer que votre application reste opérationnelle et disponible pour les utilisateurs, et vous assurer que ses performances restent acceptables et ne se dégradent pas avec le temps.

Il est important de concentrer vos efforts là où le gain sera le plus grand; dans ce but, vous devez comprendre où sont les « points chauds » de votre application. Espresso peut vous aider à le faire et ensuite vous aider pour une analyse plus étendue de ces « points chauds » pour être sûr qu'il tournent le plus vite possible.

11.1 - PERFORMANCE HOGS

Il y a un certain nombre de choses qui peuvent ralentir une application Espresso. En contrôlant attentivement cette liste, vous pouvez vérifier que vous ne travaillez pas en l'air et résoudre immédiatement des problèmes de performance.

Voici quelques éléments à regarder:

- **Logging**

Log4j, qui est le mécanisme de log intégré dans Espresso est très efficace et son impact sur les performances est très petit comparé à d'autres méthodes. Mais le log le plus rapide est pas de log du tout et un ajustement soigneux du fichier `expressoLogging.xml` et du fichier correspondant de votre application peut entraîner un bénéfice important sur la performance.

La première chose du log à regarder est bien sûr le niveau de trace de votre application. Si vous êtes encore en phase de test et de débogage , alors la priorité "debug" peut être appropriée pour au moins quelques objets, mais dans un environnement de production, vous n'avez pas besoin de plus que « info » et encore mieux « warn ».

L'un des moyens les plus faciles de voir si vous avez quelque chose de trop détaillé est de lire le fichier log et d'y chercher des messages « debug » ou « info » que vous ne voulez pas vraiment voir en production, puis de trouver pourquoi ces messages sont encore actifs et d'éditer en conséquence le fichier xml de configuration.

Un autre point important pour la performance du log est le format du paramètre « ConversionPattern » du message de log fourni à la configuration du canal de log .

Pendant les tests , il est souvent utile de voir les classes et méthodes exactes.

11.2 - CACHE TUNING

Sauf si vous n'avez pas activé cette option. (voir la documentation du fichier de propriétés) Espresso , en tournant, rassemble des informations sur l'efficacité du système de cache des objets base de données. Cette information contient le nombre d'opérations de lecture (retrieve(), searchAndRetrieve(), find(), etc) faites pour un Objet base de données particulier (dans un contexte DB donné) et le nombre de ces lectures qui pourraient utiliser le cache pour trouver l'enregistrement cherché. L'idée générale étant que plus on retrouve souvent l'enregistrement dans le cache plutôt que dans la base de données, plus l'application est rapide.

Le servlet "Status" permet de retrouver cette information, qui calcule aussi le pourcentage de « hits » de l'objet de base de données fournis par le cache ; ce que vous cherchez dans ce listing est un grand nombre de hits avec un faible pourcentage (ou 0) . Ceci indique un Objet base de données fortement utilisé par les opérations de lecture, mais peu « caché ». Le résultat de Status montre aussi la taille du cache courant pour cet objet ; si c'est zéro, vous n'avez pas d'entrée pour l'objet dans la table «Limites de page d'objet de base de données » (accédée depuis la page « Configuration » de Espresso).

Il y a d'autres facteurs à considérer pour le cache des objets de base de données :

- **Mémoire disponible**

Le cache peut utiliser pas mal de mémoire, aussi faut-il vérifier la mémoire disponible pour votre application avant de trop augmenter la taille du cache. Le servlet « Status » donne un "snapshot" de l'utilisation courante de la mémoire, mais vous pouvez consulter le log pour voir l'utilisation de la mémoire au fil du temps. Vous pouvez ajuster le niveau de détail du log pour l'objet "core.cache.CacheManager" pour voir une trace périodique. Si vous voyez que le gestionnaire de cache doit souvent réinitialiser le cache pour obtenir plus de mémoire (c-à-d, l'augmenter au delà du minimum spécifié – voir la documentation du fichier de propriétés), alors vous ne pourrez sans doute pas augmenter le cache sans ralentir l'application.

Assurez-vous de lancer votre machine virtuelle java (ou votre serveur d'applications) avec les options appropriées pour avoir assez de mémoire pour votre application – ce n'est pas parce que vous avez assez de mémoire physique sur votre système que votre JVM va l'utiliser. Voir l'option -X de Java pour savoir comment augmenter cette limite.

- **Éléments cachés**

Quelques objets de base de données peuvent montrer une forte activité et un faible cache, mais ne pas poser de problèmes. Ceci est particulièrement vrai pour les éléments qui sont *toujours* cachés par Espresso, comme les valeurs de configuration, les entrées de sécurité pour les objets DB et Contrôleurs et quelques autres.

- **Patrons écrits**

L'efficacité du cache est également influencée par les patrons par lesquels votre application écrit dans la base de données – si un objet base de données particulier est modifié, son entrée de cache est supprimée (pour que la donnée périmée ne soit pas lue) et il doit être relu dans la base de données au prochain accès. Si vous avez un objet avec un taux de cache élevé, mais guère efficace, cela peut être que cet objet est trop fréquemment modifié.

Ceci est plus du domaine du développement d'application que quelque chose directement adressable par l'optimisation de cache.

Faites attention à la manipulation des enregistrements dans la table "Database Object Page Limit", vous pouvez faire un meilleur usage de votre mémoire par le cache des objets base de données. C'est souvent l'optimisation la plus importante que vous pouvez faire.

11.3 - HEALTHCHECK ET LES TESTS DE PERFORMANCE

Les conditions qui affectent la performance d'une application peuvent changer avec le temps. Un serveur peut se trouver occupé, le réseau peut devenir lourdement chargé, une fuite de mémoire peut se manifester, etc. Il est important que la performance de votre application reste acceptable même quand les conditions changent. Expresso vous facilite cela.

☰ Test de Performance

Il y a un lien sur la page "Opération" de Expresso pour la maintenance de "tests de performance".

Ce lien permet de tester un URL donné et de spécifier un critère de performance pour cet URL. L'URL n'a pas besoin de faire partie d'une application Expresso – ce peut être n'importe quel URL dont vous voulez vérifier qu'il est opérationnel. La première fois que vous spécifiez un URL, vous pouvez ne pas savoir quelle performance raisonnable en attendre. La table Performance Test rassemble les statistiques au fur et à mesure que le test se déroule, ce qui vous permet de voir comment la performance évolue après une série de tests; donc, en effectuant des tests répétés à différents moments, les statistiques vous donneront des valeurs raisonnables pour les champs de performance.

En plus de vérifier que l'exécution de l'URL ne prend qu'un certain temps, la table de performance vous permet de contrôler que l'URL renvoie un résultat attendu. Le champ "Expect String" est utilisé pour spécifier une chaîne de caractères qui est supposée faire partie du résultat de l'URL quand il s'exécute correctement. Pour une simple page web statique, vous pouvez spécifier quelques mots du texte de la page. Pour un URL dynamique (comme un JSP ou un servlet) vous pouvez donner un terme ou deux de ce qui est attendu. Soyez sûr que le terme donné n'apparaîtra PAS si la page échoue (c-à-d pas de tags HTML ou quoi que ce soit qui puisse être une partie d'un message d'erreur). Alors le test de performance est capable de vérifier que le résultat attendu est celui reçu et de rendre compte du problème dans le cas contraire.

☰ Utiliser les tests de Performance : HealthCheck

Une fois que vous avez défini une série de tests de performance, l'utilitaire HealthCheck permet de les exécuter périodiquement. Le plus facile est d'écrire un script schell ou un fichier batch et de le soumettre au gestionnaire de tâches de votre système (cron, System Scheduler ,etc).

Par exemple, le script pour un système Linux ressemble à :

```
#!/bin/sh
java -classpath /usr/orion/orion.jar:
    /usr/orion/applications/expresso/expresso-web/WEB-INF/classes:
    /usr/orion/applications/expresso/expresso-web/WEB-INF/lib/mysql_comp.
    /usr/orion/applications/expresso/expresso-web/WEB-INF/activation.jar:
    /usr/orion/applications/expresso/expresso-web/WEB-INF/lib/mail.jar
    com.jcorporate.expresso.core.utility.HealthCheck
    configDir=/usr/orion/applications/expresso/expresso-web/config db=
```

La ligne qui commence par "java" est une seule ligne, coupée ici pour plus de clarté .

Ce script peut être exécuté, par exemple, toutes les heures via la commande "cron" en spécifiant une entrée crontab comme ceci :

```
SHELL=/bin/sh
MAILTO=root
0 * * * * healthcheck.sh
```

Quand HealthCheck s'exécute, il lit chaque test de performance défini et rassemble les statistiques de performance au cours de son exécution. Si un URL ne renvoie pas la chaîne "Expect" ou que la tâche est trop longue, il construit un email envoyé en tant que notification d'événement. Le code d'événement "HEALTH" permet de déterminer qui reçoit la notification, laquelle ressemble à :

```
From: support@javacorporate.com
Sent: Tuesday, March 27, 2001 12:01 PM
To: mnash@javacorporate.com
Subject: ERROR: System Health Check
HealthCheck
HealthCheck at 2001-03-27 08:00:11
For database/context 'default'
There were no failures, no warnings, and 1 caution,
CAUTION: Test 6 (Center for Espresso) for URL
'http://www.jcorporate.com/components/
internal/Center.jsp?category=65'
responded slower than it's set normal time of 160 milliseconds.
It ran in 237 milliseconds.
>From Server:www.jcorporate.com,
Database/context:default (Default Database)
```

L'exemple ci-dessus montre le message reçu quand un URL dépasse le temps normal , mais pas celui d'alerte tout en s'exécutant correctement ('c-à-d qu'il renvoie la chaîne spécifiée).

Un côté pratique du processus HealthCheck est qu'il renvoie un email même s'il n'y a pas d'alerte – cet email est en événement "success" , aussi vous pouvez facilement choisir de ne voir que les messages où il y a un problème, mais les événements de succès sont un bon indicateur que tout va bien sur le site.

La seule condition d'erreur que HealthCheck ne trouve pas est celle où le serveur ne marche pas du tout – empêchant HealthCheck de tourner – ou celle où le serveur email ne délivre plus ses messages. Cependant si vous savez que vous devez avoir un message toutes les heures, alors vous pouvez réagir s'il n'est pas arrivé.

11.4 - OPTIMISATION DE LA BASE DE DONNEES

Si toutes les techniques d'optimisation et de performance ci-dessus indiquent qu'un problème dans l'accès à la base de données ralentit le système (malgré le cache db) , alors, il est quelques fois nécessaire d'analyser les requêtes de base de données elles-mêmes pour déterminer laquelle est la cause du problème.

L'Objet DBConnection qui est responsable de l'exécution des requêtes SQL construites par les objets DB aide cette analyse : basculez sa priorité de log à "debug" dans le fichier expressoLogging.xml et contrôlez le résultat.

Le temps de chaque requête SQL sera inscrit, ce qui vous donnera les plus fautives et vous pourrez soit modifier votre application soit utiliser des index de base de données pour rendre les opérations plus rapides.

- ASCII** (American Standard Code for Information Interchange)
Ce jeu de caractères standard est largement utilisé dans la transmission de données.
- ANSI** (American National Standards Institute)
Ce groupe est l'organisation US membre de ISO, International Organization for Standardization.
- DTD** (Document Type Definition)
Un DTD est la définition formelle des éléments, structures et règles pour **marking up** un type donné de document SGML. Vous pouvez mettre un DTD au début d'un document ou dans fichier séparé.
- HTML** (HyperText Markup Language)
C'est le format des fichiers **publiés**. HTML est une application de SGML; pour écrire en HTML en utilisant un logiciel basé SGML, vous n'avez besoin que du DTD HTML.

ANNEXES



EXPRESSO EN TANT QU'APPLICATION FRAMEWORK

Bien qu'il soit possible d'utiliser les composants d'Expresso sans utiliser Expresso en entier, le système rend la construction d'application WEB complète plus facile quand il est utilisé en tant qu'application **framework**.

Expresso impose un minimum de contraintes à une application WEB. Il ne vous oblige à aucune méthode ou style particuliers de navigation ni même à aucune architecture particulière.

Les application WEB construites en utilisant Expresso comme **framework** peuvent utiliser JSP, des *servlets* ordinaires et même des *applets* ou des applications **standalone** comme UI.

A.1 - STRUCTURE DE L'APPLICATION

La structure fondamentale d'une application Expresso est contenue dans l'objet « **Schema** » de l'application. Cet objet est une extension de la classe « **Schema** » et sert de liste de tous les autres objets contenus dans une application donnée. Expresso lui-même utilise la classe **Schema** (com.jcorporate.core.ExpressoSchema) pour décrire les classes communes à toutes les applications Expresso.

Expresso peut alors utiliser cet objet **Schema** pour configurer automatiquement une application ; le *Servlet* `DBCCreate` lit la liste de tous les objets **Schema** connus et peut :

1. Créer toutes les tables requises par les objets base de données de l'application.
2. Fournir la sécurité de base qui permet au groupe Admin d'accéder à toutes les nouvelles tables.
3. Fournir la sécurité de base pour tous les objets Contrôleurs de l'application.
4. Fournir la sécurité de base pour tous les objets Job de l'application.

Une fois l'application initialisée , Espresso fournit la gestion de l' application :

1. Le *log* de l'application peut être consulté et géré par les services de base fournis par Espresso.
2. La sécurité de l'application est gérée dans un style Matrix commode. (par l'emploi des objets contrôleurs ControllerSecurityMatrix, DBObjSecurityMatrix, et JobSecurityMatrix).
3. Les *jobs* de l'application sont mis en file d'attente par le contrôleur de base QueueJob ; ce contrôleur peut être étendu pour les besoins de files d'attente personnalisées.
4. Tous les objets base de données de l'application sont maintenus (création, modification, suppression, recherche) par le *serv/et* de base DBMaintAuto simplement en passant le nom de la base de données ; les instructions pour le faire sont [ici](#).

FICHER HTML POUR EXPRESSO

B.1 - INTRODUCTION

Expresso utilise et génère des pages HTML (ou JSP) qui utilisent **Cascading Style Sheets** (CCS) – Niveau 1 pour le formatage actuel des pages. Ceci permet à l'apparence d'un site qui utilise Expresso d'être cohérente.

La couronne de gloire de CCS est qu'en changeant simplement quelques lignes dans un fichier, connu comme le fichier de feuille de style, l'apparence entière du site peut être modifiée.

Les sections suivantes détaillent comment créer des fichiers HTML pour Expresso, et aussi comment utiliser les classes de génération de HTML pour tirer parti des possibilités de CCS.

Il est fortement recommandé que les auteurs de HTML, JSP et servlets se retiennent de définir explicitement les polices, couleurs,... à la création et se reposent sur les possibilités fournies par CCS pour contrôler l'apparence du site

B.2 - LA FEUILLE DE STYLE

Les styles utilisés par un fichier CCS sont dans un fichier de feuille de style. Le fichier de feuille de style utilisé par Espresso se situe dans ***/components/expresso/style/default.ccs*** .

Dans le même répertoire, c-à-d /components/expresso/style sont d'autres fichiers CCS qui peuvent remplacer default.ccs pour varier l'apparence.

Le fichier ***expresso.ccs*** contient la configuration CCS par défaut et c'est d'habitude celui qui est recopié dans default.ccs.

La feuille de style default.ccs définit (quand le corps d'une page HTML utilise l'attribut class= »jc-default » ou class= »jc-toc », la couleur de fond de la page, la police par défaut et la couleur d'écriture, la couleur des liens et des liens visités. Elle définit aussi la couleur, la taille et l'alignement de divers styles comme définit ci-dessous.

L'effet du fichier default.ccs sur une page d'exemple peut se voir [ici](#)

B.3 - UN MODELE DE FICHER HTML POUR EXPRESSO

Un fichier HTML/JSP pour Expresso devrait avoir le format suivant :

```
<html> <head> <meta http-equiv="content-type" content="text/html; charset=iso-8859-1">
<title>The Page Title</title> <meta http-equiv="Content-Style-Type"
content="text/css"> <link href="/components/expresso/style/default.css"
rel="stylesheet" type="text/css"> </head> <body class="jc-default"> <p class="jc-
pageheader">The Page Header</p> </body> </html>
```

Trois choses sont à noter :

1. Dans la section <head>...</head>, est référencée la feuille de style default.ccs
2. Le corps contient l'attribut **class= »jc-default »**
3. L'en-tête de page est défini comme un paragraphe avec l'attribut **class= »jc-pageheader »**

B.4 - TABLES HTML POUR EXPRESSO

La feuille de style default.ccs devrait être définie avec les en-têtes de table encadrés par `<th>...</th>` at l'attribut ***class= »jc-tabletitle »***.

Voici un exemple :

```
<table align="center" border="1" cellspacing="2" cellpadding="0"> <tr> <th class="jc-tabletitle">Heading 1</th> <th class="jc-tabletitle">Heading 2</th> <th class="jc-tabletitle">Heading n</th> </tr> <tr> <td>Data 1</td> <td>Data 2</td> <td>Data 3</td> </tr> </table>
```

B.5 - AUTRES STYLES

La feuille de style default.ccs définit un certain nombre d'autres styles utilisables par les diverses pages. Ce sont :

Style	Description
jc-successheader	En-tête de page quand une opération réussie est affichée
Jc-errorheader	En-tête de page quand une condition d'erreur est affichée
Jc-description	Pour un texte de description (d'habitude un texte statique)
Jc-info	Pour un texte d'information (d'habitude après une action)
Jc-explanation	Pour un texte d'explication (d'habitude un texte statique)
Jc-instruction	Pour un texte informant l'utilisateur de ce qu'il doit faire (d'habitude sur une fiche)
Jc-label	Etiquettes de champs sur les fiches
Jc-reqlabel	Etiquettes de champs obligatoires sur les fiches
Jc-formfield	Ligne de table qui contient des champs
Jc-formnote	Description de champ sur les fiches

B.6 - GENERATION DE HTML AVEC EXPRESSO

L'ensemble des classes u paquet ***com.javacorporate.common.html*** est utilisé pour la génération de sortie HTML.

Toutes les classes d'élément HTML ont une méthode ***display(PrintWriter)***, laquelle génère du HTML CCS.

La plupart des classes d'élément HTML sont héritées de la classe de base `HtmlElement`. La classe (abstraite) `HtmlElement` définit deux méthodes ***setCCSClass(String)*** et ***setCSSID(String)*** *that aid with CCS information*.

La méthode `setCCSClass(String)` est utilisée pour définir le style affiché en utilisant l'attribut « class= ». Les CCS ID permettent de remplacer les attributs de style CCS pour une classe particulière, en définissant les attributs associés à cet ID seul. Les CCS ID sont spécifiés par `setCSSID(String)`.

B.7 - UNE PAGE EXEMPLE

La génération de HTML dans Expresso commende par la classe Page. Quand la méthode `display(PrintWriter)` est invoquée sur un objet Page, la sortie HTML automatiquement générée référence la feuille de style `default.ccs`.

Voici comment une page typique est générée (pour d'autres exemples, voir les méthode `doGet(...)` des classes `com.jcorporate.expresso.core.servlet` et `com.jcorporate.expresso.core.servlet`).

```
..... PrintWriter out = response.getWriter();
..... Page myPage = new Page("An example page");
..... Paragraph myPara = new Paragraph(new Text("This is the header"));
..... myPara.setCSSClass("jc-pageheader");
// Could use jc-successheader or jc-errorheader myPage.add(myPara);
..... myPage.display(out);
.....
```

B.8 - UN EXEMPLE DE TABLE

Voici un extrait du *ServletLogin* :

```
..... Form myForm = new Form("login form");
..... myForm.setAction(response.encodeUrl(getServletPrefix("Login").toString()));
..... myPage.add(myForm);
..... Table formTable = new Table("login form table");
..... myForm.add(formTable);
..... Row oneRow = new Row("user name row");
..... Cell oneCell = new Cell(new Text("User Name: "));
..... oneCell.setCSSClass("jc-label");
// Could use jc-reqlabel for required labels oneRow.add(oneCell);
..... oneCell = new Cell(new TextField("UserName", userNameDefault, 30, 30));
..... oneCell.setCSSClass("jc-formfield");
..... oneRow.add(oneCell);
..... formTable.add(oneRow);
.....
```

B.9 - AJOUTER DES CELLULES D'EN-TÊTE AUX TABLES

Il y a deux façons d'ajouter des cellules d'en-tête à une table. La première est d'utiliser une cellule régulière comme ci-dessus pour chaque cellule d'en-tête, puis de positionner les attributs suivants :

```
..... Cell oneCell = new Cell(new Text("Header"));
..... oneCell.setHeaderCell(true);
..... oneCell.setCSSClass("jc-tabletitle");
.....
```

La seconde est d'ajouter toutes les cellules d'en-tête en une fois :

```
..... myTable.setTitle("Header 1|Header 2|Header n");
//Uses "|" character as separator .....
// In above example addHeading(String) can be used interchangeably with
..... setTitle(String)
```

Enfin un titre est ajouté à une table par :

```
..... myTable.setCaption("This is the caption");
.....
```



ORGANISATION DU PAQUET D'EXPRESSO

Ce document décrit l'organisation du paquet du code source d'Expresso et la structure de fichiers de l'application web Expresso.

Les deux structures ont subi une réorganisation pour les rendre plus logiques et faciles à comprendre. Cette structure ne s'applique à Expresso 3.0 ou plus. Si vous voulez convertir depuis une version plus ancienne, voir la [documentation sur l'outil de changement de nom](#).

Veillez vous référer à JavaDoc pour les détails les plus récents de la structure.

Tous les paquets Expresso sont dans « com.jcorporate.expresso ».

Dedans, il y a 4 sous-catégories principales :

1. **com.jcorporate.expresso.core**

La classe core est requise par Presque toutes les applications Expresso comme la persistance DB, la sécurité, la machine d'état, etc. Dans ce paquet : ExpressoSchema. L'objet Schema primaire d'Expresso.

Cet objet Schema enregistre tous les autres objets obligatoires d'Expresso

Sous-paquets :

- a. **cache** Le but général du Cache Manager, qui gère tous les caches mémoires d'Expresso
- b. **controller** La définition du noyau des objets Contrôleur d'expresso

- c. **db** Les définitions du pool de connexion aux bases de données et des connexions aux bases.
- d. **Dbobj** Les définition du noyau des objets base de données.
- e. **Job** Les définitions de la classe job pour la prise en charge des tâches de fond et des jobs.
- f. **Jsdkapi** Les objets pour supporter plusieurs versions de l'API servlet.
- g. **Logging** L'interface d'Expresso pour le paquet de logging log4j et ceux qui alimentent le log de base de données
- h. **Misc** Divers objets, dont le ConfigManager qui regroupe toutes les propriétés de configuration d'Expresso et est responsable du lancement du système.
- i. **Security** Le sous-système de sécurité, y compris les liens pour l'encryptage fort et faible.
- j. **Servlet** Les servlets de base d'Expresso, dont DBMaint.
- k. **Utility** Programmes utilitaires indépendants, c-à-d qui contiennent une méthode main et sont conçus pour se lancer de la ligne de commande. Cela comprend ReNameUtil, DBTool, JobHandler et d'autres.

2. com.jcorporate.expresso.services

Des services supplémentaires font encore partie de la base de Expresso, mais ne sont pas, à strictement parler, nécessaires à une application Expresso. Ainsi du déchargement, de la vérification email, etc.

Ces sous-paquets comprennent :

- a. **controller** Les objets Contrôleur utilisés pour gérer Expresso, la file des jobs, contrôler le CacheManager et autres fonctions communes.
- b. **Dbobj** Les objets de base de données utilisés dans les services essentiels de Expresso, comme la sécurité, la mise en file d'attente, le logging.
- c. **Html** Un simple paquet de génération HTML utilisé par les servlets de base. Voir les objets Contrôleur pour des options UI plus sophistiquées.
- d. **Job** Quelques objets job de base
- e. **Servlet** Les servlets de base de Expresso, dont DBCreate, Status et d'autres
- f. **Test** Les définitions de la structure de base de l'unité test de Expresso. En cours d'être intégré à Junit.

3. **com.jcorporate.expresso.ext**

Les extensions qui seront utilisées par quelques applications Espresso, mais non par toutes. Ce sera l'endroit où mettre les futurs **bells and whistles**.

Quelques sous-paquets ne pourront pas se compiler sur certains systèmes ; c-à-d le sous-paquet «Ldap» réclament les paquets d'extension jdi et ldap (qui peuvent être téléchargés depuis Sun).

Ces paquets peuvent être ignorés sans inconvénient dans les systèmes qui ne les utilisent pas.

- a. **dbobj** Définitions de l'Objet base de données pour les services d'extension comme le déchargement, le log de déchargement, l'auto-enregistrement des utilisateurs.
- b. **Job** Les jobs pour les services d'extension comme les notifications envoyées par email aux utilisateurs qui téléchargent certains fichiers.
- c. **Ldap** Les objets d'intégration LDAP pour lire l'information de sécurité à partir d'un serveur LDAP- exige JNDI
- d. **Report** Extension de reports, dont l'analyse des utilisateurs qui ont téléchargé des fichiers.
- e. **Servlet** Extension de servlets tels un servlet de redémarrage (pour relire les valeurs de configuration), des objets DB d'exportation en format XML et un servlet pour exécuter les requêtes SQL arbitraires.

4. **com.jcorporate.expresso.demo**

Ces classes servent à montrer des fonctionnalités d'Espresso comme la connexion/déconnexion, l'enregistrement d'utilisateurs, etc. Elles ont pour but de servir d'exemple sur la façon d'utiliser Espresso.

Il est possible de supprimer les classes de ce paquet sans compromettre aucune des fonctionnalités d'Espresso.

Le sous-paquet «demo» est une application Espresso complète et a sa propre classe Schema, com.jcorporate.demo.DemoSchema.

L'ORGANISATION CVS D'EXPRESSO

L'organisation des fichiers Expresso dans CVS est faite pour permettre à Expresso de se mettre facilement en paquets dans un format de fichier .EAR (Application Entreprise) ou .WAR (Archive d'application WEB). Ce format permet à Expresso d'être immédiatement déployé sur les serveurs qui le supportent.

Si vous contrôlez l'archive CVS de Expresso dans le répertoire de nom expresso, vous verrez une structure de ce genre :

- ❖ `expresso/` ce sera le répertoire de niveau 1 si vous mettez Expresso dans un fichier .ear
 - **META-INF/** répertoire obligatoire pour la structure Entreprise Application.
 - **Application.xml** fichier de définition obligatoire pour la structure Entreprise Application.
 - **Expresso-ejb** non encore utilisé. Contiendra la définition de **bean** de l'entreprise (EJB) utilisée par l'application
 - **Expresso-web** répertoire de niveau 1 pour Expresso en tant qu'application web.
 - `Frame.jsp`. le fichier « bienvenue » ou fichier initial de Expresso. C'est vers ce fichier que votre navigateur s'oriente quand vous entrez pour la première fois dans cette application web.
 - `Toc.jsp`
 - `Applications.jsp`. La page utilisée pour « enregistrer » les applications Expresso
 - `Error.jsp` . la page standard d'erreur pour Expresso et les applications Expresso
 - `Header.jsp`. trame d'en-tête de page

- LICENSE. Fichier qui détaille la licence à laquelle est reliée Expresso et les licences d'autres paquets tiers intégrés dans Expresso
- Main.jsp
- Operation.jsp La page d'opérations de Expresso
- README
- SUPPORT. Le fichier qui explique les procédure de support et la possibilité de payer une prime de support.
- Config/ Ce répertoire est le répertoire initial de configuration, pointé par web.xml dans le sous-répertoire WEB-INF décrit ci-dessous . Il est recommandé de le déplacer à un endroit plus sûr pour un déploiement en production.
 - Default.properties. c'est le fichier de propriété par défaut de Expresso, configuré au départ pour utiliser la base de données fournie Hypersonic. Voir la [documentation du fichier de propriété](#) pour plus de détails
 - Log4j.xml . Ce fichier configure le sous-système de log log4j intégré dans Expresso. Voir la [documentation du log](#) pour plus de détails
- Db/ Fichiers d'exemple de base de données SQL Hypersonic avec des tables Expresso déjà créées
- Doc/ . Documentation Expresso, dont ce document fait partie.
- Help/ Fichiers d'aide au format HTML
- Images/ Fichiers des images par défaut
- Javadoc/ Documentation javadoc pour Expresso
- Jsp/ divers fichiers JSP utilisés dans l'administration d'Expresso
- Log/ Par défaut, les fichiers log sont situés dans ce répertoire. Il est recommandé d'éditer log4j.xml pour spécifier une situation plus sûre
- Style/ La feuille de style par défaut. (default.ccs) est ici.
- Components/ ce sous-répertoire contient un répertoire pour chaque application Expresso (cad, eForum, ePoll, eSearch, etc)
 - Eforum/
 - Epoll/
 - Esearch/
 - Demo/

- WEB-INF/ Ce répertoire spécifié pour la structure .war, contient les classes java d'Expresso et toutes les autres classes requises par cette application web.
 - Lib/ Ce répertoire est là où tout fichier .jar est placé pour être utilisé par l'application. C'est là où il faut placer les fichiers qui contiennent les pilotes JDBC pour toutes les autres bases de données autre que celle fournie Hypersonic
 - Classes/
 - Com/
 - Jcorporate/
 - Expresso/ Tous les fichiers .class et .java de Expresso. Les sous-répertoires correspondent à l'organisation du paquet décrite ci-dessus. Remarquez qu CVS ne contient que des fichiers .java, mais que les distributions téléchargeables d'Expresso contiennent à la fois des fichiers .class et des fichiers .java
 - Esearch/ tous les fichiers .class et .java pour esearch
 - Eforum/ Tous les fichiers .class et .java pour eforum
 - Epoll/ tous les fichiers .class et .java pour epoll
 - Web.xml. Ce fichier, défini par la spécification d'application web , contient l'information qui définit cette application web, dont l'enregistrement de tous les servlets.



LES ASTUCES POUR SECURISER UN ENVIRONNEMENT EXPRESSO DE PRODUCTION

Expresso offre beaucoup de fonctionnalités de sécurité pour de solides options de **déploiement**. Cependant, les administrateurs système doivent garder à l'esprit la règle numéro un de la sécurité :

La sécurité est un **processus**

Ce qui veut dire que toute application serveur doit subir un **audit de routine** et qu'il y a toujours des sécurités supplémentaires disponibles. Des modifications, audits et **patches** des procédures sont toujours nécessaires. Si une société prétend avoir une sécurité impénétrable, « à l'épreuve des balles »,... *elle ment!*

Voilà le début d'une liste de contrôle pour vous aider à sécuriser l'environnement Expresso .

- **Faites tourner votre application sous la version la plus récente possible de JVM** – Il y a toujours des corrections de bogues dans les nouvelles révisions et certains bogues peuvent être exploités par des *hackers*.
- **Exécuter la configuration de Expresso avec une police de sécurité java** – Comme la version 1.2 de JDK, Java a quelques possibilités très fines de sécurité. Voici ce qu'il a au minimum à faire avec la police :
 - *Limiter les classes qui accèdent au fichier système aux seules indispensables. Ce sont principalement les classes de configuration et ne devrait être que celles-ci.*

- *Limiter les classes qui peuvent se connecter au réseau.* Vous avez aussi la possibilité de déterminer de quelle adresse i vous autorisez une connexion. C'est parfait quand vous faites tourner votre application serveur sous la dépendance d'un autre serveur web (voir [Apache Jakarta projet](#)). En gros, vous ne voulez que les connexions directement issues du serveur web. Vous voudrez également vous assurer que seules les classes client JDBC et celles responsables des connexions sont autorisées à effectuer des connexions externes. De cette façon, un *hcker* ne peut pas ajouter de petite classe qui envoie les mots de passe directement sur son site.

Voir la documentation JDK pour plus d'informations sur la création des fichiers police.

- **Sécuriser le fichier système.** Assurez-vous que les seuls utilisateurs qui ont accès aux répertoires de l'application serveur sont «root» pour un contrôle complet et le compte sous lequel va tourner l'application pour les seules permissions d'accès de lecture et d'exécution. Si vous avez des répertoires dans lesquels l'application serveur va écrire des fichiers, donnez l'accès à ces répertoires, mais souvenez-vous que là où un fichier peut être écrit, un *hacker* peut s'introduire. Donc, limitez les permissions d'accès aux fichiers système. Assurez-vous également qui vous avez complètement restreint l'accès au répertoire de configuration de Espresso.
- **Sécurisez les pilotes de base de données JDBC.** Selon sur quelle machine est située la base de donnée, l'étape « Exécuter sous une police de sécurité » peut changer. Assurez-vous que les pilotes DB n'acceptent que les connexions de sources données et bloquez toutes les autres tentatives de connexion.
- **Employez des coupe-feux entre internet et votre base de donnée.** C'est ennuyeux quand votre serveur web devient indisponible... Il s'agit d'**un problème légal** si votre base de données de cartes de crédit en entier est lue par un *hacker*... Protégez votre base de données par tous les moyens possibles.
- **Faites tourner la base de données Espresso avec des privilèges restreints.** Quoiqu'Espresso ait un bon contrôle sur les enregistrements que les utilisateurs peuvent ajouter, modifier ou supprimer, il est cependant possible de contourner le fichier de configuration de base de données de Espresso, d'entrer directement dans la base de données et de taper ce genre de commande :

**DROP TABLE CUSTOMER_ORDERS ;
COMMIT;**

Aussi, vérifiez que les seules permissions accordées aux applications Espresso sont celles absolument indispensables.

- **Enlevez tous les services non indispensables.** Quelques *servlets* et services peuvent ne pas être utilisés dans un environnement de production. Par exemple, le *servlet* RunSQL ne sera sans doute pas utilisé directement. Ou votre application n'utilise pas la fonctionnalité de sauvegarde de fichier. Enlevez de votre serveur d'application chaque service qui n'est pas obligatoire au site web pour fonctionner.
- **Enlevez tous les mots de passe par défaut.** Le projet [Openhack](#) a été « craqué » simplement parce qu'un utilisateur a oublié de changer un mot de passe par défaut venu avec Oracle. Assurez-vous de changer tous les mots de passe des comptes utilisateurs par défaut. Ou encore mieux, créez un compte administratif sous un nom différent et supprimez tous les comptes utilisateurs par défaut. C'est de là qu'un *hacker* doit commencer.
- **Exécuter des applications de monitoring de sécurité.** Les plus déterminés des *hackers* sont éventuellement entrés. Ce n'est qu'une question de temps. Il vaut mieux avoir une série de systèmes de détection d'intrusion en place. Ainsi, vous serez alertés en cas d'activité suspecte. Ou, si vous en avez les moyens, achetez-vous le service de monitoring qu'offre [Counterpane Internet Security](#).
- **Exécuter des audits externes.** Il est presque impossible de voir ses propres fautes. Ayez une (ou deux) interventions externes pour réaliser au moins un audit de base de votre système serveur web. Cela devrait au moins corriger quelques unes des erreurs les plus courantes faites lors de l'installation.
- **Vérifiez que la bibliothèque est sérieusement cryptée.** Par défaut, cette option est activée, et à moins que ce ne soit absolument nécessaire, ne la désactivez pas.
- **N'utilisez pas Windows 95 ou Windows 98 comme serveur !!!** Je suis désolé que cela doive même figurer ici, mais beaucoup de gens s'imaginent que Personal Web Server / Win98 est suffisamment sécurisé pour une utilisation de production. C'EST COMPLETEMENT FAUX ! N'importe qui peut obtenir le contrôle complet d'un système Win95/98 aussi vite que vous diriez « Qu'est-ce qui se passe ? » Si vous devez utiliser windows comme serveur, à tout le moins, prenez Windows 2000 ou encore mieux NT4 avec Service Pack 6. [Je sais que NT4 est assez vieux, mais MS a ajouté un certain nombre de fonctionnalités **non testées dans Win2K qui conserveront une sécurité raisonnable jusqu'à Service Pack 2 ou 3.** Quand il y aura assez de Service Packs au-delà... allez-y et abandonnez NT4]

Je répète que la sécurité sera toujours un processus. Vous devrez toujours modifier quelque chose dans votre serveur pour le rendre plus sûr, même si ce n'est qu'appliquer les derniers patches de sécurité disponibles auprès de diverses sources. Ce document devrait vous donner un point départ décent du travail à réaliser quand on utilise un environnement de production sécurisé. Si vous n'êtes pas familier de la création de machines de production avec une sécurité de qualité, cherchez sur internet des trucs du genre « Security FAQ » et intéressez-vous aux livres sur la sécurité pour avoir une meilleure idée de ce dont vous devez vous protéger et comment le faire au mieux.

Si un problème survient à l'exécution d'Expresso ou d'une application développée avec Expresso, plusieurs voies d'action peuvent aider à déterminer la cause du problème. Vérifiez les éléments suivants dans l'ordre où ils apparaissent :

1. Le moteur servlet/JSP

1. Vérifiez que le **moteur** de servlet et servlet-serveur tourne et sur quel port. Souvent les **moteurs** de servlet n'utilisent pas le port par défaut 80, mais peut-être le port 8080, 7602, etc. Assurez-vous aussi que le **moteur** servlet peut servir des fichiers HTML **statis** à travers le même port. Vous pouvez vérifier sa capacité à le faire en regardant des fichiers comme http://votrenomdeserveur:votrenumerodeport/components/expresso/doc/propriete_s.html , en adaptant le nom du serveur et le numéro de port.
2. Les **moteurs** de servlet viennent souvent avec des servlets et pages JSP exemples ; soyez sûr que ces exemples tournent correctement avant de vous mettre à déboguer Expresso.

2. Fichier de propriétés

Ensuite, il faut vérifier que le fichier de propriétés qui donne l'information initiale à Expresso et celle de connexion à la base de données est correct et peut être lu correctement.

1. Vérifiez que vous avez au moins le fichiers « default.properties » dans votre répertoire de configuration. Quand d'autres fichiers de propriétés sont permis, celui-ci au moins doit exister.
2. Il ne doit pas y avoir d'autres fichiers que ceux de propriétés dans le répertoire de configuration. Détruisez tous les autres fichiers ou déplacez-les.

3. Vérifiez que votre fichier de propriétés est bien lu quand le moteur de servlet démarre. Le moyen de dire à un moteur de servlet de le faire varie d'un moteur à l'autre, mais vous pouvez vérifier les choses suivantes pour vous assurer que cela se passe bien :
 - Regardez la sortie standard (d'habitude un fichier log) de votre moteur de servlet. Vous devriez voir des lignes qui disent que le fichier de propriété a été lu, que le pool de connexion à la base de données est initialisé, ainsi de suite. Si vous ne voyez pas ces messages, DefaultInit peut ne pas s'exécuter au démarrage.
 - Exécuter de le servlet InitServletProps (avec l'URL <http://votreserveur:votrenumerodeport/servlet/InitServletProps>). Il affichera les propriétés lues par le servlet DefaultInit pendant le démarrage du serveur.
 - Exécuter le servlet Status (avec l'URL <http://votreserveur:votrenumerodeport/servlet/Status>). Il vous informe sur le pool de connexion et le type **mappings** de la base de données appliqué à votre installation. Si status ne s'exécute pas, cela indique probablement que DefaultInit n'a pas démarré proprement ou que l'information de votre fichier de propriétés est incorrecte.

3. Connexion de base de données

Si Espresso a lu sa configuration correctement, mais ne s'exécute pas correctement, il pourrait être incapable de se connecter à votre base de données. Voici les étapes que vous pouvez suivre pour vous assurer que votre base de données est bien connectée :

1. Exécuter le servlet Status. Il vous informe sur le pool de connexion et le type **mappings** de la base de données.
2. Exécuter JobHandler en redirigeant les messages vers un fichier. JobHandler prend un argument configDir=xxx, où xxx est le même répertoire de configuration que celui passé au servlet DefaultInit. Si JobHandler peut se connecter, mais non vos servlets alors le problème est **sans doute** dans le servlet DefaultInit ou **classpath** de votre moteur de servlet. Souvenez-vous que JobHandler peut utiliser, pour tester les connexions aux bases de données, autre chose que la valeur par défaut avec l'argument db=xxx, où xxx est la première partie du nom du fichier dans le répertoire configDir ; c-à-d pour utiliser « oracle.properties », tapez db=oracle.
3. Une autre application JDBC. Si vous avez une autre application qui peut ~~exec~~ applicJDBC. Si vopliet Defa 5 TD -0.15cliet De0723 Tc351es,

Oracle a SQL Plus, etc. Soyez sûr que le nom de la base de données référencé dans votre fichier de propriétés existe déjà ou créez-le et réessayez.

5. Servlet DBCreate. Si vous n'êtes pas certain que toutes les tables de votre application existent, il est toujours plus sûr d'exécuter le servlet DBCreate. Il créera toutes les tables manquantes et vous dira aussi si votre base de données est accessible.
6. DBTool: le programme **standalone** DBTool peut être utilisé (avec l'option « verify ») pour contrôler une base de données existante. Verify contrôlera l'intégrité référentielle (que les enregistrements liés ne sont pas détruits ou modifiés) et que toutes les tables nécessaires existent bien ainsi que tous leurs champs.

4. Valeurs de configuration

Des erreurs sont également dues aux valeurs incorrectes de configuration. Les valeurs de configuration sont semblables aux valeurs des fichiers de propriétés, mais sont spécifiques à une application et une base de données particulières et sont stockées dans la base de données elle-même. La signification exacte de chaque valeur dépend de l'application à configurer; il faut vérifier dans la documentation de l'application et s'assurer que chaque valeur est correcte.

5. Pages HTML/JSP

Il est important de vérifier que les pages HTML/JSP pour Espresso sont correctement installées à un endroit approprié. Par défaut, (c-à-d à moins que vous ne le changiez) les pages sont installées dans un répertoire appelé « /components » dans le répertoire racine web de votre serveur web/servlet. Elles devraient être servies par le même **moteur** et sur le même port. Cad, si vous accédez au servlet Status par `http://votreserveur:8080/servlet/Status`, alors la page web de l'installation d'Espresso doit être `http://votreserveur:8080/components/espresso/installation.html`

Vérifiez que les pages JSP sont opérationnelles sur votre serveur par la page **jsp Login**, accessible par `http://votreserveur:votrenumerodeport/espresso/jsp/login.jsp` . Si cette page ne travaille pas correctement, vérifiez que le **processeur** de pages JSP est activé sur votre **moteur** de servlet et que le support de JSP 1.0 ou plus est positionné (c-à-d pas JSP 0.9 qui est une option de quelques serveurs de servlet).

6. Configuration de l'application

Une fois que vous avez vérifié tout le fonctionnement d'Expresso, vous devez vérifier que les étapes supplémentaires pour rendre votre application opérationnelle ont été suivies :

1. Enregistrements des servlets supplémentaires
2. **Mise en liste** de l'application dans la fonction Schema Listing (sur la page Expresso « applications » à <http://votreserveur:votrenumerodeport/components/expresso/application.jsp>)
3. Exécuter le servlet DBCreate pour créer les tables de l'application et autres valeurs de configuration.
4. **Mettez** la sécurité des fonctions de l'application.
5. Toutes les valeurs de configuration sont renseignées pour l'application.

☰ Erreurs de servlet

Quand un servlet envoie un message d'erreur, le message est soit un message formaté Expresso (en-tête rouge «Erreur de servlet » et une table qui contient les détails) soit il vient du serveur de servlet sous-jacent. Une autre sorte d'erreur indique un problème spécifique à l'application ou une erreur de configuration dans le moteur de servlet lui-même.



ERREURS DE SERVLET EXPRESSO

Les erreurs de servlet Expresso ont une forme constante : La boîte d'erreur affiche toujours trois choses :

1. L'objet de l'erreur :

Il indique le nom de l'objet et celui de la méthode de l'objet qui a provoqué l'erreur. Cela permet de tracer ce qui déclenche le message d'erreur.

2. Numéro de message :

Quand une erreur a un numéro associé, les détails de l'erreur sont accessibles en cliquant sur le lien du numéro de message. S'il n'y a pas de numéro, il n'y a pas de lien.

3. Message d'erreur :

Le texte du message. Il contient aussi le message d'erreur généré par le moteur de base de données si l'erreur est relative à un problème de base de données.

Davantage d'information est contenue dans cette sorte de page d'erreur, à l'intention des développeurs et des administrateurs système : voir le source HTML de la page d'erreur et, si c'est possible, une trace de la pile java encadrée dans un commentaire vers la fin du fichier HTML. Si cette trace n'a pas de numéro de ligne, désactivez (temporairement) tout JIT (compilateur Just-in-time) que votre machine virtuelle java peut utiliser pour voir ces numéros de ligne. Cette information devrait vous permettre de trouver directement la ligne de code qui génère l'erreur.

☰ Erreurs de serveur de servlet

Les messages d'erreur qui ne sont pas du format ci-dessus sont souvent générées par le moteur de servlet lui-même. Cela peut indiquer un problème avec la configuration du serveur de servlet ou un problème avec les fichiers de propriétés des servlets Expresso.

Un message de la forme «Cannot load servlet:xxx » indique habituellement que la ligne qui référence ce servlet n'existe pas ou n'indique pas un chemin valide pour le fichier servlet .class. Voir les informations d'installation pour plus de détails.

☰ Logging

Quand une erreur survient, l'information détaillée est aussi envoyée à l'objet Expresso Log. L'objet Log écrit dans plusieurs sources, selon ce qu'il a de disponible :

- a. Le log HTML ; Le nom de fichier de ce log est indiqué dans les valeurs de configuration définies pendant l'installation. La base de données doit être disponible pour utiliser ce genre de log.
- b. Le servlet log : La plupart des moteurs de servlet fournissent un fichier log dans lequel sont écrits tous les appels d'un servlet « log ».
- c. Sortie standard : La plupart des moteurs de servlet fournissent aussi un fichier log pour écrire les données tracées dans System.out. Le log écrira aussi dans ce fichier si c'est possible.

Quand on cherche une information log, Il faut contrôler chacune de ces possibilités et le log System.err fourni par le moteur de servlet.

Il est souvent profitable d'augmenter temporairement le niveau de détail du log pour résoudre un problème.

☰ Trace

Essayez de mettre à « y » une propriété du fichier de configuration de nom « trace ». Cela tracera toutes les opérations DBConnectionPool et DBConnection sur la sortie d'erreur standard du serveur. Cela peut être très utile si vous obtenez des résultats que vous n'attendiez pas et si vous voulez voir quel SQL est exactement exécuté par la base de données.

Si vous avez tout essayé et n'avez toujours pas trouvé le problème, veuillez nous envoyer un message à support@javacorporate.com, ou consultez les archives à opensource@servlet.net.

Vous trouverez aussi une liste des bogues connus sur la page Developers de notre site web <http://www.javacorporate.com>



VALEURS DE CONFIGURATION

Une fois que Espresso est initialisé, vous pouvez utiliser le servlet de maintenance de base de données pour voir et modifier la liste des valeurs de configuration.

Setup Code	Description
AdminEmail	L'adresse email du contact d'administration/support. Cette devrait être renseignée avec l'adresse email d'un utilisateur qui recevra les requêtes de support du site web ; elle est utilisée à la construction de l'email « verify » pour l'authentification utilisateur basée email.
AdminName	Le nom du contact d'administration/support du webmaster du site. C'est le nom à utiliser pour l'adresse email spécifiée ci-dessus
BaseDir	Répertoire racine de documents web du serveur. Cela devrait être le répertoire de la racine de votre serveur web.
CompanyName	Nom de la société ; cette valeur est aussi utilisée pour la vérification email quand l'authentification utilisateur basée email est utilisée.
ConnTimeOut	L'intervalle de timeout en secondes pour la connexion de base de donnée ; après ce délai, la connexion est considérée perdue et est retournée au pool de connexion.
ContextPath	Le chemin du contexte d'Espresso. Il s'agit d'une valeur très importante pour que tous les composants d4expresso fonctionnent correctement. Par défaut, Espresso est installé dans le répertoire /components/expresso (relatif au répertoire racine web), mais avoir une autre valeur selon votre installation. Pour les installations de servlet API 2.2 , c'est Context Path pour le contexte d'application d'Espresso.
DefaultGroup	Groupe d'utilisateur par défaut pour les utilisateurs auto-enregistrés ; cette valeur fournit le code d'un groupe de sécurité. Les utilisateurs qui se servent de la fonction d'auto-enregistrement d'Espresso sont automatiquement membres de ce groupe.
EmailValidateURL	URL complet du servlet de validation email ; c'est l'URL à utiliser dans l'authentification email des utilisateurs quand elle est activée.
HomePageURL	URL de la page d'accueil . C'est l'URL incluse dans l'authentification email si elle est activée.

HTTPServ	Nom de l'hôte du serveur web ; c'est le nom du système hôte du serveur de l'application
LogLevel	Niveau de détail du log (0=minimum,9=maximum ; vous pouvez l'initialiser à la valeur maximale (9) et réduire ensuite quand le système est en production.
MAILFrom	Valeur du champ From dans les emails d'événement ; ce doit être une adresse email valide sur un serveur d'email spécifié pour que les envois d'email d'événement soit opérationnels
MAILPassword	Le mot de passe pour envoyer un email au besoin ; si votre serveur d'email demande un nom d'utilisateur et un mot de passe pour l'envoi des emails d'événement, il doit être mis ici.
MAILServer	Le nom du serveur SMTP pour l'envoi d'emails ; le nom d'hôte du serveur qui abrite votre serveur d'emails. Espresso utilise ce serveur pour l'envoi de toutes les notifications email.
MAILUserName	Le nom d'utilisateur pour l'envoi d'email au besoin ; Si le serveur d'email demande un nom d'utilisateur et un mot de passe, ce champ renseigne le nom d'utilisateur.
MaxConnections	Le nombre maximum de connexions autorisées à la base de données ; le pool de connexion déclenchera une exception si davantage de connexions sont demandées au même moment. Cela est très pratique quand une limite externe est imposée au nombre de connexions que la base de données peut supporter. S'il n'y a pas de limites, cette valeur peut être assez haute (20 ou 30 est raisonnable).
MaxJobs	Le nombre maximum de serveurs de jobs qui peuvent s'exécuter en même temps ; cela limite le nombre de requêtes JobHandler exécutées en parallèle. Mettre 1 indique une file de job mono-tâche.
RequireEmailValidate	Validation email obligatoire pour l'activation de compte ; Mettre Y pour activer le processus d'authentification email d'utilisateurs.
RestartServer	Ligne de commande pour relancer le serveur ; une ligne de commande sur le serveur qui réinitialisera le moteur de servlet, utilisée par le processus HealthCheck si le moteur de servlet échoue (en expérimentation).
ServletEvent	Mettre ce champ à Y si vous voulez recevoir des emails pour toute exception rencontrée par les utilisateurs ou applications Espresso. Un événement SYSERROR sera déclenché et l'email contiendra une trace complète de l'erreur et d'autres détails. Mettre « N » si vous ne voulez recevoir ces messages email, et mettre « E » générera des envois pour toutes les exceptions sauf pour les exceptions de sécurité.
SerletPath	Le chemin par défaut des servles ; il s'agit du préfixe des servlets dans Espresso, par défaut /servlet. Pour les installation de servlets API 2.2 (comme J2EE) c'est le chemin du contexte de l'application Espresso
ServletPort	Numéro de port du serveur de servlet ; le numéro de port http pour utiliser les requêtes de servlet. La valeur par défaut est souvent 80 quand aucun numéro de port n'est fourni dans les URLs
TempDir	Répertoire temporaire ; un répertoire temporaire que Espresso peut utiliser pour stocker des fichiers sur le serveur. Il faut un nom de chemin absolu su terminant par un « / ».

TimerInterval	Tous les combien la file des jobs et les répertoires sont-ils analysés ? La file de jobs est analysée tous les tant de secondes pour rechercher de nouvelles requêtes de job.
---------------	---

Les valeurs de configuration prennent effet dès qu'elles sont enregistrées.



DESCRIPTION DE FICHIERS DE PROPRIETE

Expresso a besoin d'au moins un fichier de "propriétés" pour se connecter à votre base de données. Un fichier de propriétés est un fichier de texte avec des paires nom/valeur spécifiées qui donnent à Expresso l'information de base à propos de votre base de données et celle nécessaire pour initialiser un pool de connexion pour se connecter à la base de données.

Les fichiers de propriétés sont lus par le servlet DefaultInit (sauf si vous implémentez votre propre servlet d'initialisation) quand le moteur de servlet démarre. Les fichiers de propriétés doivent être dans leur propre répertoire sans aucun autre fichier, et le nom du répertoire est passé en argument d'initialisation (de nom "configDir") au servlet DefaultInit par le moteur de servlet.

Le premier fichier de configuration doit s'appeler "default.properties". Les connexions pour d'autres bases de données peuvent s'appeler comme vous voulez. Dans ces fichiers, on doit trouver ce genre de texte:

```
dbDriver=org.hsql.jdbcDriver
dbURL=jdbc:HypersonicSQL:/usr/db/expresso
dbConnectFormat=3
dbLogin=sa
dbPassword=
images=/components/images
dbCache=y
loginForm=/components/expresso/jsp/login.jsp
dbTest=select 0
description=Hypersonic Default Database
dbMap.LONGVARCHAR=LONGVARCHAR
```

Voici un [lien vers un fichier de propriétés de base](#) que vous pouvez enregistrer et modifier.

Pour une installation de base, vous ne devez changer qu'une seule information dans le fichier ci-dessus; là où la chaîne /usr/db/expresso est en gras.

Cette chaîne de caractères doit devenir le chemin d'un répertoire utilisé par Hypersonic pour stocker ses fichiers de base de données.

Sous Windows, vous pouvez préfixer le nom du répertoire par une lettre de lecteur, comme d:/data/expresso.

Le répertoire doit exister et être vide au départ.

Le processus DBCreate de Espresso génèrera pour vous toutes les tables et valeurs nécessaires.

Nous vous recommandons de commencer par utiliser la base de données incluse Hypersonic, puis de vous connecter ensuite à la base de données de votre choix, quand vous aurez installé et configuré Espresso proprement.

Propriété	Signification	Obligatoire?	Valeur exemple
description	Une description de l'environnement et de la connexion décrite par ce fichier de propriétés. Affichée par le servlet Login quand il demande de choisir un environnement.	Non	Base de données MySQL
dbDriver	Nom d'objet du pilote JDBC utilisé pour se connecter à la base de données	Oui	org.gjt.mm.mysql.Driver
dbURL	URL à utiliser avec le pilote donné pour se connecter à la base spécifiée.	Oui	jdbc:mysql://javacorp.com/javacorp
dbConnectFormat	Chois de l'un des 3 "formats" possibles pour la connexion. Les formats possibles sont décrits dans une table séparée ci-dessous.	Oui	2
dbLogin	Nom de connexion fourni à la base de données.	Oui	test
dbPassword	Mot de passe fourni à la base.	Oui	test
images	Répertoire utilisé pour accéder aux images utilisées par l'auto-génération de pages HTML. Vous ne devez *pas* mettre le slash terminal du nom de répertoire.	Oui	/components/images
trace	Ecrit les messages de trace sur la sortie standard. La sortie standard est normalement dirigée vers un fichier log par le moteur de servlet, ce qui est très utile pour déboguer.	Non, "N" par défaut	y
loginForm	Cette propriété, si elle est spécifiée, autorise tout message "Accès refusé" à présenter une fiche de connexion personnalisée après le message qui permet à l'utilisateur d'essayer de se connecter ou de changer sa connexion. Si elle n'est pas spécifiée, alors les messages "Accès refusé" seront simplement affichés.	Non	/components/expresso/jsp/login.jsp
dateSelectFormat	Cette propriété spécifie le format dans lequel votre DBMS fournit les valeurs des champs "date". Si vous spécifiez une dateSelectFunction, alors dateSelectFormat doit être le format APRES l'application de dateSelectFunction. Vous pouvez laisser cette propriété en dehors de votre fichier de config (ou la spécifier à blanc)	Non, aaaa-MM-jj par défaut.	yyyy-MM-dd

	<p>et la valeur par défaut sera utilisée.</p> <p>Les "patterns" autorisés pour spécifier un format sont les mêmes que ceux permis par le constructeur de la classe <code>java.text.SimpleDateFormat</code>. Voir la Javadoc JDK standard.</p>		
<code>dateSelectFunction</code>	<p>Cette propriété peut spécifier une fonction spécifique de la base de données à utiliser pour les champs de données DATE. Elle doit retourner la valeur date dans le format spécifié par <code>dateSelectFormat</code>. Laisser à blanc pour n'appliquer aucune fonction (mais simplement prendre le champ date comme il est fourni par la base). Toute occurrence de %s dans la chaîne sera remplacée par le nom du champ traité - par exemple, si vous voulez <code>DATEFORMAT(StartDate, 'yyyy-MM-dd')</code>, vous devez spécifier la propriété <code>dateSelectFunction=DATEFORMAT(%s, 'yyyy-MM-dd')</code>.</p>	Non	<code>DATEFORMAT(%s, 'yyyy-MM-dd')</code>
<code>dateUpdateFunction</code>	<p>Cette propriété peut spécifier une fonction de la base à utiliser pour modifier les données d'un champ DATE. Laisser à blanc pour n'appliquer aucune fonction (mais simplement prendre le champ date comme il est fourni par <code>dateUpdateFormat</code>). Toute occurrence de %s dans la chaîne sera remplacée par la valeur du champ traité - par exemple, si vous voulez <code>DATE('2001-01-01', 'yyyy-MM-dd')</code>, vous devez spécifier la propriété <code>dateUpdateFunction=DATE(%s, 'yyyy-MM-dd')</code>.</p>	Non	<code>DATE(%s, 'yyyy-MM-dd')</code>
<code>dateUpdateFormat</code>	<p>Ce champs spécifie le format des colonnes "date" en modification - cad, pour les instructions SQL insert et update. Pour quelques bases de données (en particulier MySQL) les formats update et select sont différents.</p>	Non, yyyy-MM-dd par défaut	<code>yyyy-MM-dd</code>
<code>timeSelectFormat</code>	<p>Même chose que <code>dateSelectFormat</code>, mais pour les champs "time"</p>	Non, HH:mm:ss par défaut	<code>HH:mm:ss</code>
<code>timeSelectFunction</code>	<p>Même chose que <code>dateSelectFunction</code>, mais pour les champs "time"</p>	Non	
<code>timeUpdateFunction</code>	<p>Exactement comme <code>dateUpdateFunction</code>, mais appliqué seulement aux champs de type TIME. Laisser à blanc pour n'appliquer aucune fonction</p>	Non	
<code>timeUpdateFormat</code>	<p>Même chose que <code>dateUpdateFormat</code>, mais pour les champs "time"</p>	Non, HH:mm:ss par défaut	<code>HH:mm:ss</code>
<code>dateTimeSelectFormat</code>	<p>Même chose que <code>dateSelectFormat</code>, mais pour les champs "datetime".</p>	Non, yyyy-MM-dd HH:mm:ss par défaut	<code>yyyy-MM-dd HH:mm:ss</code>
<code>dateTimeSelectFunction</code>	<p>Même chose que <code>dateSelectFunction</code>. mais pour</p>	Non	

	les champs "datetime".		
dateTimeUpdateFormat	Même chose que dateUpadteFormat, mais pour les champs "datetime".	Non, yyyy-MM-dd HH:mm:ss par défaut	yyyy-MM-dd HH:mm:ss
dateTimeUpdateFunction	Exactement comme dateUpdateFunction, mais appliqué seulement aux champs de type DATETIME. Laisser à blanc pour n'appliquer aucune fonction	Non	
dbWildCard.n	Cette propriété permet de remplacer les caractères de substitution par défaut de la base de données par d'autres plus appropriés à votre DBMS particulier. Voir la description détaillée et la table de valeurs exemples ci-dessous.	Non	dbWildCard.1=% dbWildCard.2=*
startJobHandler	Cette propriété permet de dire à Expresso de démarrer son JobHandler comme une tâche identique au moteur de servlet VM. Ceci élimine la nécessité de faire tourner JobHandler dans un processus séparé du serveur. Les valeurs possibles de cette propriété sont "y" ou "n". Ne pas mettre la propriété dans le fichier est la même chose que de la spécifier à "n".	Non	startJobHandler=y
strongCrypto	Cette propriété permet de sélectionner le support de cryptage fort de Expresso - Activer cette fonctionnalité nécessite soit les extensions de cryptage Java de Sun soit l'implémentation de Cryptix JCE. En outre, sous JDK1.2 ou plus, le fichier jre/lib/security/java.security doit être configuré pour indiquer quel logiciel de cryptage est utilisé. Si cette valeur n'est pas définie, le cryptage fort est actif par défaut , et le JCE Cryptix ou Sun doit être dans votre CLASSPATH. Mettre strongCrypto=n pour utiliser les logiciels de cryptage "faible" (encore plus sécurisants qu'avant!) qui ne réclament pas JCE.	Non	strongCrypto=n
createTableIndicies	Cette propriété permet de désactiver la création des index de table pendant le processus de création de la base de données (DBTool ou DBCreate). Mettre cette propriété à "n" pour empêcher la création d'index si pour une raison quelconque vous ne voulez pas créer d'index sur les tables de votre base.	Non	Par défaut, il y aura création d'index
cacheManager.minMemoryPercentage	Cette propriété renseigne le pourcentage minimum de mémoire autorisé par le gestionnaire de cache - si le pourcentage de mémoire libre (cad la mémoire libre prise comme pourcentage de mémoire totale) descend en-dessous de cette valeur, le gestionnaire de cache commencera à diminuer les caches pour essayer de libérer	Non	cacheManager.minMemoryPercentage=10

	assez de mémoire. Si vous ne voulez pas que le gestionnaire de cache le fasse, mettez la valeur à 0. Par défaut, cette valeur est 10. Les décimales sont autorisées(e.g. 5.5)		
cacheManager.maxRetries	Quand le gestionnaire de cache détecte que la mémoire disponible est tombée en dessous du seuil indiqué ci-dessus, il entame un cycle de recherche des caches possibles à supprimer pour retrouver assez de mémoire. Cette propriété permet de spécifier combien de fois le cycle a lieu-une limite par défaut est fournie pour s'assurer que le cycle se termine.	Non	cacheManager.maxRetries=30
language	Décide de la langue par défaut (voir java.util.Locale) pour ce contexte. Cette langue est utilisée si aucune langue n'est spécifiée comme préférence utilisateur pour l'utilisateur courant (ou si l'utilisateur courant ne peut pas être déterminé) et si le navigateur de l'utilisateur ne spécifie pas de préférence de langue (ou s'il n'y a pas de navigateur).	Non	language=en
country	Donne la pays par défaut (voir java.util.Locale) de ce contexte. Ce pays est utilisé si aucun pays n'est spécifié comme préférence utilisateur pour l'utilisateur courant (ou si l'utilisateur courant ne peut pas être déterminé) et si le navigateur de l'utilisateur ne spécifie pas de préférence de pays (ou s'il n'y a pas de navigateur).	Non	country=us
styleSheet	Donne l'emplacement des feuilles de style des applications Expresso. S'il n'est pas spécifié, l'emplacement des feuilles de style sera /style/default.css, à l'endroit désigné par la valeur de "ContextPath" dans le schéma (Expresso) de base. Notez que la valeur de styleSheet est un URL, nom pas un nom de fichier. Vous ne devez pas utiliser un URL relatif (cad, il doit commencer par "/").	Non	styleSheet=/components/expresso/style/default.css
mailDebug	ce flag, quand il est mis à "y", active une information détaillée de débogage de l'objet EMailSender vers la console. Ce flag peut être utilisé pour aider à résoudre des problèmes sur la capacité d'Expresso à envoyer des emails.	Non	n
errorHandler	Cette propriété fournit le nom de l'objet HTTPController à utiliser comme objet de gestionnaire d'erreur dans ce contexte. Notez que si une erreur survient alors que le contexte ne peut pas être déterminé, alors le gestionnaire d'erreur "par défaut" est utilisé. Voir ci-dessous .	Non	com.jcorporate.expresso.services.controller.ErrorHandler
errorHandlerParams	Paramètres supplémentaires en option pour le contrôleur d'erreurs. Ils peuvent être	Non	pas de paramètres

	<p>utilisés pour personnaliser l'affichage (par exemple) d'un contexte donné. Cad que mettre la valeur</p> <p>"next!%context%/jsp/showerror.jsp" utilisera la page "showerror.jsp" pour afficher la sortie du gestionnaire d'erreur, d'où un affichage plus élaboré des erreurs. La page "showerror" peut bien sûr être copiée et modifiée à volonté.</p>		
dbStats	<p>Cette option permet d'arrêter les statistiques de cache de la base de données. Cette collection de statistiques est utilisée par le servlet "Status" pour produire une table montrant les accès aux objets de base de données et leurs ratios de lecture dans le cache mémoire. Cette information peut être utilisée pour déterminer quels objets db de votre application a été utilisée le plus souvent et ajuster le cache en accord avec une performance maximum. La collection de ces informations est nécessaire, cependant, coûte une peu de performance, aussi ce flag existe pour pouvoir désactiver les statistiques et peut être utilisé pour les systèmes optimisés de production pour empêcher le minimum de surcharge occasionnée</p>	Non	Statistics collection is ON
preFlushHTTPResponse	<p>Quelques servlets peuvent prendre un certain temps pour s'exécuter, ce qui provoque un timeout sur la connexion HTTP et l'affichage d'une page erreur par le navigateur. Si la propriété est mise à "y", les méthodes doGet/dopost d'un servlet (dérivé de StdServlet) feront un flush() de la sortie.</p>	Non	preFlushHTTPResponse=y

H.1 – VALEURS DE DBConnectFormat

Les valeurs de DBConnectFormat permettent à Expresso d'utiliser différentes sortes de chaînes de connexion dans le même code, ce qui autorise à se connecter à n'importe quelle source de données JDBC sans avoir à personnaliser le code.

Les valeurs possibles de DBConnectFormat sont :

DBConnectFormat	Signification
1	DriverManager.getConnection(dbURL, login, password) où login et password sont le nom de connexion et le mot de passe fournis
2	Crée la connexion par un appel à DriverManager.getConnection(dbURL + "?user=" + login + ";password=" + password) où login et password sont le nom de connexion et le mot de passe fournis
3	Crée la connexion en mettant le nom et le mot de passe dans un objet Properties et en appelant DriverManager.getConnection(dbURL, props) où props est l'objet properties créé
4	Crée la connexion par une chaîne comme DriverManager.getConnection(dbURL + "?user=" + new Login + "&password=" + new Password)

Vous pouvez déterminer quel DBConnectFormat utiliser en vous référant à la documentation de configuration de votre pilote JDBC.

Si vous ne le pouvez pas, il est assez simple d'essayer tous les formats et de voir lequel fournit une connexion correcte à votre base.

H.2 – VALEURS DE dbWildCard

Il y avait un grand hasard potentiel dans les versions précédentes d'Expresso en ce qui concerne les caractères de substitution de base de données.

Le problème pouvait se produire, par exemple, quand un utilisateur était enregistré avec un souligné dans son nom de connexion, ex: "my_login".

Si vous essayiez de retrouver cette connexion (disons par l'écran de l'utilisateur admin), vous obteniez un erreur db selon laquelle aucun enregistrement n'était trouvé... même si vous saviez que l'enregistrement existait.

Le problème venait de ce que les quatre caractères suivants étaient définis comme caractères de substitution par défaut dans les versions précédentes d'Expresso: "%", "[", "]", "_" .

Dans l'exemple ci-dessus, le nom d'utilisateur contenait un "_", aussi, le SQL était formé avec un "select * from userlogin where username LIKE 'my_login'";. Cela ne retourne rien dans PostgreSQL, par exemple. "_" n'est pas un caractère de substitution dans PostgreSQL.

Pour corriger cela, DBConnectionPool a été modifié, en ajoutant une nouvelle méthode appelée setWildCards().

Quand le pool est initialisé, si ConfigManager trouve des clés préfixées par "dbWildCard", ces valeurs sont utilisées A LA PLACE des quatre caractères par défaut. Ainsi, si un seul caractère de substitution est défini dans le fichier de propriétés, c'est le seul qui sera utilisé par l'application quand elle testera des caractères de substitution.

Voici à quoi doivent ressembler les entrées du fichier de propriétés:

```
# Database WildCard Characters
dbWildCard.1=%
dbWildCard.2=*
- etc
```

Noter la numérotation 1,2,... pour avoir des clés uniques.

Nous prions les utilisateurs Expresso de nous faire part de leur sentiment sur ce sujet pour de futures versions d'Expresso. Les quatre caractères par défaut peuvent provoquer des résultats inattendus si vous continuez à les utiliser.

Cependant, si vous n'ajoutez pas d'entrées à vos fichiers de propriétés, la fonctionnalité d'Expresso ne changera pas, aussi il n'y a pas lieu de s'inquiéter de cette modification et de casser les applications existantes.

Database	Valeurs dbWildCard
PostgreSQL	dbWildCard.1=%

☰ Correspondance de types de base de données

Les messages d'erreur qui ne sont pas du format ci-dessus sont souvent générées par le moteur de servlet lui-même. Cela peut indiquer un problème avec la configuration du serveur de servlet ou un problème avec les fichiers de propriétés des servlets Expresso.

Depuis la version 1.05, Expresso supporte une flexibilité complète de correspondance de types de base de données.

Si vous installez Expresso pour la première fois, vous utiliserez simplement la correspondance par défaut qui sera adéquate pour beaucoup de systèmes de base de données, auquel cas vous ne spécifierez aucune des propriétés ci-dessous.

Quand vous configurez un objet de base de données, vous spécifiez un type pour chacun des champs de l'objet - les noms que vous utilisez pour ceci sont "internes" à Expresso, et ne sont pas nécessairement ceux utilisés dans la base de données. Cad, vous pouvez dire "text", mais votre base de données enregistre les champs "text" sous un type de nom "VARCHAR(4000)", par exemple.

Ce que la correspondance de type fait est d'établir deux correspondances: La première est du nom de type "interne" à Expresso vers les types standards JDBC de java.sql. Cette correspondance a des valeurs par défaut qui fournissent une correspondance qui convient à la plupart des bases de données. Vous pouvez écraser ces valeurs ou en ajouter en mettant une entrée dans le fichier de propriétés (au même endroit où vous spécifiez les paramètres de connexion) qui ressemble à ceci :

```
expressoType.foo=CHAR
```

Cela définit un nouveau type Expresso, appelé "foo", qui correspond au type interne java.sql.Types.CHAR . Ainsi, vous pouvez alors dire quelque chose comme: addField("UserName", "foo", 30, false, "User Name"); dans la définition d'un objet bases de données.

Il y a ensuite une seconde correspondance : des types java.sql vers les noms de type utilisés par votre base de données. D'habitude le pilote JDBC que vous utilisez fournit une correspondance appropriée, via les méta-données DB, mais cela ne se produit pas toujours ou pas toujours correctement. Pour le pilote MySQL que nous utilisons en interne, la méta-donnée ne renvoie rien du tout, par exemple.

Le système fait alors ce qui suit: Il construit une correspondance par défaut des types java.sql vers les types de base de données, puis soit l'écrase ou y ajoute la correspondance spécifiée par le pilote JDBC. Ensuite il écrase ou ajoute selon les correspondances personnalisées que vous spécifiez par des entrées comme celle-ci:

```
dbMap.LONGVARCHAR=varchar(4000)
```

cela dit à votre système que les éléments qui utilisent le type java.sql LONGVARCHAR doivent être stockés avec un type de base de données "varchar(4000)".

Et donc, si vous ne changez la correspondance par défaut d'Expresso, vous direz dans l'objet de base de données:

```
addField("Comment", "text", 0, false, "User Comment");
```

Ce qui dit de créer la colonne "Comment" en utilisant le type Expresso "text". "Text" correspond au type java.sql LONGVARCHAR. L'entrée dbMap ci-dessus dit alors d'utiliser le type de base de données "varchar(4000)" pour tous les LONGVARCHAR. Donc, quand vous exécutez le servlet DBCreate et générez les tables qui contiennent le champ "Comment", l'instruction de création dira :

```
Comment varchar(4000) null
```

Le servlet "Status" a été amélioré pour afficher la correspondance de types dans une table, et il est ainsi facile de voir les correspondances établies.

Une fois que le fichier de propriétés est créé et placé dans le répertoire approprié, vous pouvez continuer par l'étape d'initialisation dans la [Setup page](#) .

☰ Gestion d'erreur

Toute exception soulevée pendant l'exécution d'Expresso est prise en charge par un unique objet, le Contrôleur "error handler".

Par défaut, il s'agit de l'objet `om.jcorporate.expresso.services.controller.ErrorHandler`, mais peut être personnalisé en donnant à la propriété "errorHandler" un autre nom d'objet.

Quand une erreur se produit, l'objet nommé est instancié avec toute l'information disponible sur l'erreur. Le contrôleur passe ensuite à l'état "handle". Dans le gestionnaire d'erreur par défaut, cet état "handle" essaie de faire plusieurs choses:

1. tracer l'erreur dans le log `log4j`
2. Générer un événement `SYSEERROR` pour l'erreur (si cela est activé dans la configuration) et envoyer les emails appropriés
3. Si l'erreur provient d'un servlet ou d'un JSP, essayer de notifier le problème à l'utilisateur. La façon dont cela est réalisé dépend des paramètres du gestionnaire d'erreur. Vous pouvez fournir des paramètres au gestionnaire d'erreur par la propriété `errorHandlerParam`. Le paramètre le plus utile pour le gestionnaire par défaut est "next", qui spécifie un URL auquel transférer le contrôle pour afficher l'erreur. Si aucun "next" n'est fourni le gestionnaire par défaut affichera un simple message d'erreur très semblable aux messages d'erreur affichés par Expresso dans les versions précédentes (cad courts et plutôt obscurs). Utiliser une valeur de "errorHandlerParam" de "next|`%context%/jsp/showerror.jsp`" montrera un exemple de formatage de sortie d'erreur via un JSP, et le résultat sera des messages d'erreur beaucoup plus agréables pour l'utilisateur. Pour fournir plus d'un paramètre au contrôleur gestionnaire d'erreur, utilisez des paires de valeurs séparées par pipe dans la valeur de `errorHandlerParam`. Par exemple: "next|`%context%/jsp/showerror.jsp`|back|`%context%/frame.jsp`" fournira le paramètre "next" avec la valeur "`%context%/jsp/showerror.jsp`" et le paramètre "back" avec la valeur "`%context%/frame.jsp`". (Le pattern `%context%` sera automatiquement remplacé par le chemin de contexte courant).

Des paramètres supplémentaires peuvent être passés au gestionnaire d'erreur.

Ils peuvent être utilisés pour personnaliser l'affichage en sortie (par exemple) pour un contexte spécifique.

C-à-d, mettre cette propriété à la valeur "next|`%context%/jsp/showerror.jsp`" utilisera la page JSP "showerror.jsp" pour afficher la sortie du gestionnaire d'erreur.

La page "showerror" peut bien sûr être copiée et modifiée pour personnaliser l'affichage des erreurs.

☰ Fichiers de propriétés exemples

Voici quelques fichiers de propriétés dus à la contribution de notre communauté:

[Interbase DBMS](#)

[Enhydra's InstantDB](#)

[Oracle 8](#)

[Hypersonic SQL](#)