



Security Analysis of Core J2EE Patterns Project

(traduction partielle, Régis Lécu, 24 octobre 2016)

TABLE DES MATIÈRES

Introduction	3
Patterns de la couche de présentation	4
Intercepting Filter	4
Front Controller	7
Context Object	11
Application Controller	14
View Helper	17
Composite View	19
Service to Worker	21
Dispatcher View	23
Patterns de la couche métier	25
Business Delegate	26
Service Locator	28
Session Façade	30
Application Service	32
Business Object	33
Composite Entity	34
Transfer Object	36
Transfer Object Assembler	37
Value List Handler	39
Patterns de la couche intégration	40
Data Access Object	40
Service Activator	42
Domain Store	44
Web Service Broker	46
References	48



INTRODUCTION

La plupart des experts sécurité ne retiennent qu'une seule activité pour la phase de conception : la modélisation des menaces. C'est une excellente technique pour évaluer la surface d'attaque d'une application, mais d'après notre expérience, les développeurs n'ont parfois pas le temps, le budget ou les connaissances en sécurité pour construire un modèle des menaces approprié. De plus, les développeurs ne peuvent créer un modèle complet des menaces avant de terminer la conception de leur application.

Ce guide de référence a pour but de dispenser les meilleures pratiques de sécurité aux développeurs, pour qu'ils prennent des décisions de sécurité pendant la conception. Nous nous centrons sur un des concepts les plus importants dans l'ingénierie actuelle du logiciel : les patterns de conception (*design patterns*). Ils offrent un vocabulaire commun pour discuter de la conception d'une application, indépendamment des détails d'implémentation. Une des collections de patterns les mieux accueillies par la communauté des développeurs JAVA JEE est le livre *Core J2EE Patterns book* de Deepak Alur etc. Les développeurs implémentent régulièrement des patterns comme *Application Controller*, *Data Access Object* ou *Session Façade*, dans de grandes applications JEE réparties, ou dans des *frameworks* comme *Spring* ou *Struts*. Notre objectif est de dispenser les meilleures pratiques de sécurité, pour que les développeurs puissent introduire des propriétés de sécurité et éviter les vulnérabilités, indépendamment de leurs choix technologiques, tel que le choix d'un *framework* MVC (*Model View Controller*).

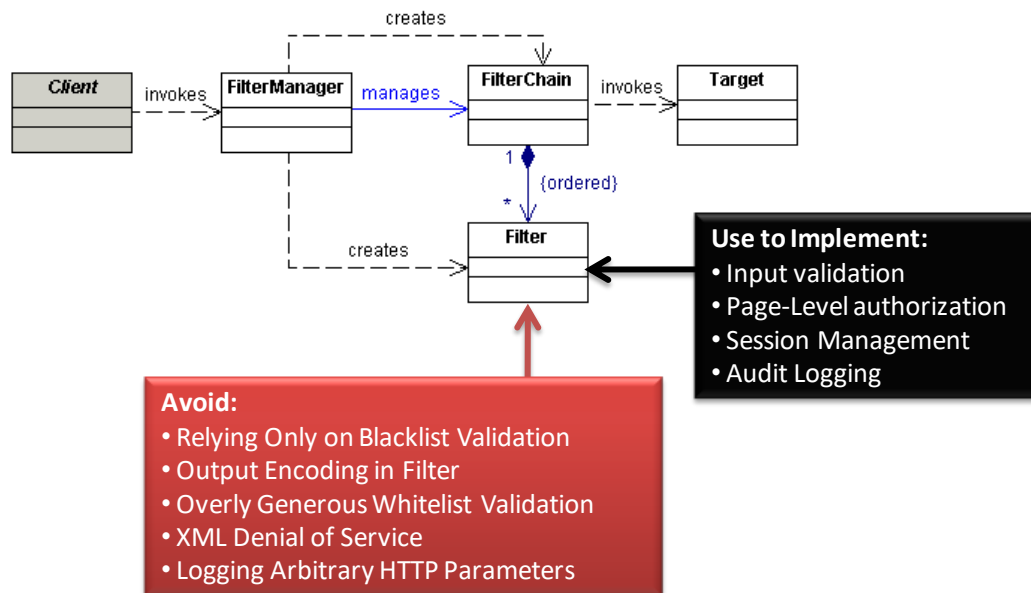


PATTERNS DE LA COUCHE DE PRESENTATION

INTERCEPTING FILTER

The *Intercepting Filter* pattern may be used in instances where there is the need to execute logic before and after the main processing of a request (pre and postprocessing). The logic resides in `Filter` objects and typically consist of code that is common across multiple requests. The Servlet 2.3 Specification provides a mechanism for building filters and chaining of `Filters` through configuration. A `FilterManager` controls the execution of a number of loosely-coupled `Filters` (referred to as a `FilterChain`), each of which performs a specific action. This [Standard Filter Strategy](#) can also be replaced by a [Custom Filter Strategy](#) which replaces the Servlet Specification's object wrapping with a custom implementation.

DIAGRAM



ANALYSIS

Avoid

Relying Only on a Blacklist Validation Filter

Developers often use blacklists in `Filters` as their only line of defense against input attacks such as Cross Site Scripting (XSS). Attackers constantly circumvent blacklists because of errors in canonicalization and character encodingⁱ. In order to sufficiently protect applications, do not rely on a blacklist validation filter as the sole means of protection; also validate input with strict whitelists on all input and/or encode data at every sink.



Output Encoding in Filter

Encoding data before forwarding requests to the `Target` is too early because the data is too far from the sink point and may actually end up in several sink points, each requiring a different form of encoding. For instance, suppose an application uses a client-supplied e-mail address in a Structured Query Language (SQL) query, a Lightweight Directory Access Protocol (LDAP) lookup, and within a Hyper Text Markup Language (HTML) page. SQL, LDAP, and HTML are all different sinks and each requires a unique form of encoding. It may be impossible to encode input at the `Filter` for all three sink types without breaking functionality. On the other hand, performing encoding after the `Target` returns data is too late since data will have already reached the sink by the time it reaches the `Filter`.

Overly Generous Whitelist Validation

While attempting to implement whitelist validation, developers often allow a large range of characters that may include potentially malicious characters. For example, some developers will allow all printable ASCII characters which contain malicious XSS and SQL injection characters such as less than signs and semi-colons. If your whitelists are not sufficiently restrictive, perform additional encoding at each data sink.

XML Denial of Service

If you use *Intercepting Filter* to preprocess XML messages, then remember that attackers may try many different Denial of Service (DOS) attacks on XML parsers and validators. Ensure either the web server, application server, or the first `Filter` on the chain performs a sanity check on the *size* of the XML message **prior** to XML parsing or validation to prevent DOS conditions.

Logging Arbitrary HTTP Parameters

A common cross-cutting application security concern is logging and monitoring of user actions. Although an *Intercepting Filter* is ideally situated to log incoming requests, avoid logging entire HTTP requests. HTTP requests contain user-supplied parameters which often include confidential data such as passwords, credit card numbers and personally identifiable information (PII) such as an address. Logging confidential data or PII may be in violation of privacy and/or security regulations.

Use to Implement

Input Validation

Use an *Intercepting Filter* to implement security input validation consistently across all presentation tier pages including both Servlets and JSPs. The `Filter`'s position between the client and the front/application controllers make it an ideal location for a blacklist against all input. Ideally, developers should always employ whitelist validation rather than blacklist validation; however, in practice developers often select blacklist validation due to the difficulty in creating whitelists. In cases where blacklist validation is used, ensure that additional encoding is performed at each data sink (e.g. HTML and JavaScript encoding).



Page-Level Authorization

Use an *Intercepting Filter* to examine requests from the client to ensure that a user is authorized to access a particular page. Centralizing authorization checks removes the burden of including explicit page-level authorization deeper in the application. The Spring Security frameworkⁱⁱ employs an *Intercepting Filter* for authorization.

Remember that page-level authorization is only one component of a complete authorization scheme. Perform authorization at the command level if you use `Command` objects, the parameter level such as HTTP request parameters, and at the business logic level such as *Business Delegate* or *Session Façade*. Remember to propagate user access control information such as users' roles to other design layers like the *Application Controller*. The OWASP Enterprise Security Application Programming Interface (ESAPI)ⁱⁱⁱ uses `ThreadLocal`^{iv} objects to maintain user authorization data throughout the life of a thread.

Session Management

Session management is usually one of the first security controls that an application applies to a request. Aside from container-managed session management controls such as idle timeout and invalidation, some applications implement controls such as fixed session timeout, session rotation and session-IP correlation through proprietary code. Use an *Intercepting Filter* to apply the additional session management controls before each request is processed.

Invalidating the current session token and assigning a new session token after authentication is a common defense against session fixation attacks. This control can also be handled in an *Intercepting Filter* specifically configured to intercept authentication requests. You may alternatively use a generic session management `Filter` that intercepts all requests, and then use conditional logic to check, specifically, for authentication requests in order to apply a defense against session fixation attacks. Be aware, however, that using a generic `Filter` introduces maintenance overhead when you implement new authentication paths.

Audit Logging

Since *Intercepting Filters* are often designed to intercept all requests, they are ideally situated to perform logging of user actions for auditing purposes. Consider implementing a `Filter` that intercepts all requests and logs information such as:

- Username for authenticated requests
- Timestamp of request
- Resource requested
- Response type such as success, error, etc.

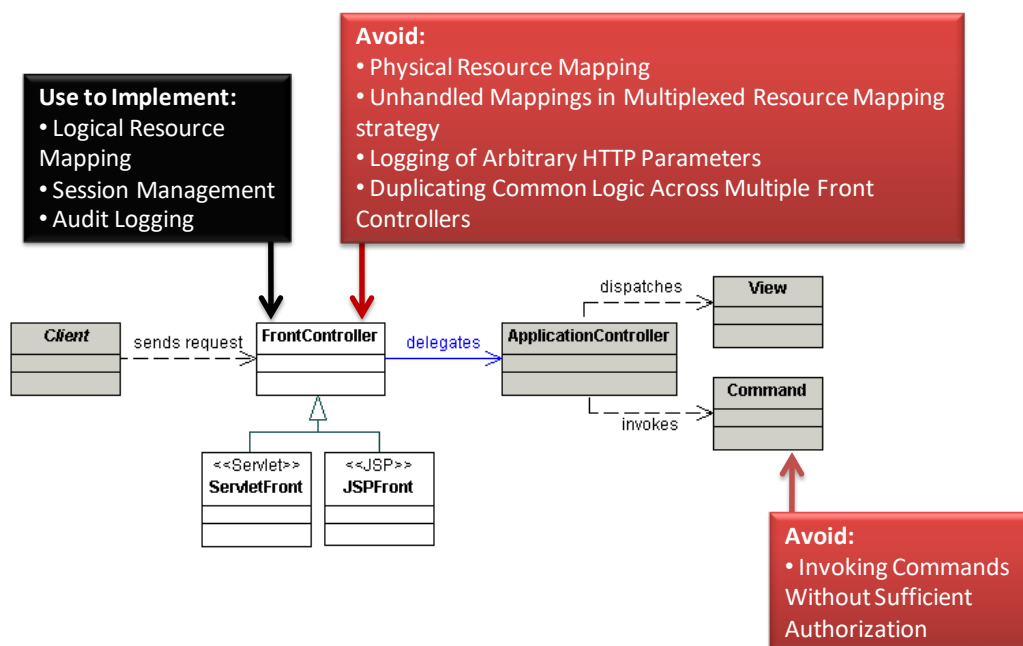
The logging filter should be configured as the first `Filter` in the chain in order to log all requests irrespective of any errors that may occur in `Filters` further down the chain. Never log confidential or PII data.



FRONT CONTROLLER

Traiter une requête comprend habituellement plusieurs étapes : prise en compte du protocole, navigation, traitement proprement dit, branchement ou construction de la vue suivante. Un contrôleur permet de centraliser la logique commune effectuée pour chaque requête. Habituellement implémenté comme une *servlet*, le *Front Controller* peut réaliser cette logique commune et déléguer ensuite les activités de gestion des actions (effectuer le traitement demandé par la requête) et de gestion des vues (envoyer une vue à l'utilisateur) à un *Application Controller*. Ce pattern permet de centraliser la logique de contrôle des requêtes et de diviser l'application entre une partie contrôle et une partie traitement.

DIAGRAMME



ANALYSE

A éviter

Mapper des ressources physiques

Cette stratégie associe directement des ressources physiques comme des fichiers situés sur le serveur, aux paramètres fournis par l'utilisateur. Les attaquants tirent souvent profit de cette stratégie pour obtenir des accès illicites aux ressources. Par exemple, dans une « exploitation par traversée de répertoire » (*directory traversal exploit*), les clients donnent au serveur l'emplacement physique d'un fichier : « *file=statement_060609.pdf* ». Les attaquants peuvent alors accéder à d'autres fichiers sur le serveur en fournissant des paramètres malveillants tels que « *file=../../../../../etc/password* ». Si l'application accepte aveuglément n'importe quel nom de fichier fourni par l'utilisateur et tente de



l'ouvrir, l'attaquant aura accès à de nombreux fichiers critiques, y compris des fichiers de propriétés et de configuration qui contiennent des mots de passe en dur.

Les développeurs limitent parfois les attaques par traversée de répertoires en testant la présence d'un préfixe ou d'un suffixe spécifique : par exemple, en vérifiant que le nom de fichier passé en paramètre commence par *statement* et se termine par *.pdf*. Mais un utilisateur astucieux peut injecter un caractère NULL (%00) en entrant « *file=statement_060609.pdf/../../etc/password%00.pdf* ». Java verra que la ressource commence par *statement* et se termine par *.pdf*, tandis que le système d'exploitation éliminera tous les caractères restant après %00 et ouvrira le fichier de mots de passe.

Fixez-vous comme règle d'éviter la stratégie de mapping des ressources physiques. Si vous y êtes obligés, assurez-vous que l'application travaille dans un environnement sécurisé (*sandbox*) avec le *Java Security Manager* et/ou faites faire suffisamment de contrôles au système d'exploitation pour protéger les ressources contre des accès non autorisés.

Lancer des commandes sans autorisation suffisante

Dans la stratégie *Command and Controller*, les utilisateurs fournissent un objet *Command* que l'objet *Application Controller* va ensuite utiliser pour effectuer une action. Les développeurs qui se reposent sur les contrôles côté client ou au niveau des pages Web, oublient souvent de tester si l'utilisateur a réellement le droit d'appeler un objet *Command* donné.

Les attaquants tirent profit de cette vulnérabilité en modifiant simplement un paramètre.

Un exemple courant est une transaction CRUD (*Create Read Update Delete*), comme <http://siteurl/controller?command=viewUser&userName=jsmith>.

Il suffit à un attaquant de changer *viewUser* en *deleteUser*. Les développeurs soutiennent souvent que si les clients ne peuvent pas voir le lien à *deleteUser* dans la page web, ils ne pourront pas appeler la commande *deleteUser*. Nous aimons nommer cela une « Autorisation basée sur l'interface utilisateur » (*GUI-based Authorization*) et c'est une vulnérabilité extrêmement courante dans les applications web.

Assurez-vous que les clients sont réellement autorisés à appeler la commande fournie, en testant leur autorisation sur le serveur d'application. Fournissez à l'objet *Application Controller* suffisamment d'informations sur l'utilisateur courant, comme son rôle et son nom d'utilisateur, pour qu'il puisse vérifier l'autorisation. Utilisez un objet *Context* pour stocker l'information utilisateur.

Unhandled Mappings in the Multiplexed Resource Mapping Strategy

La stratégie *Multiplexed Resource Mapping* associe des ensembles de requêtes logiques aux ressources physiques. Par exemple, toutes les requêtes qui se terminent par le suffixe *.ctrl* sont gérées par l'objet *Controller*. Les développeurs oublient souvent de prendre en compte les mappings inexistants, tels que des suffixes non associés à des gestionnaires (*handlers*) spécifiques.



Créez un `Controller` par défaut pour les mappings inexistants. Assurez-vous que le `Controller` fournit simplement un message d'erreur générique ; s'appuyer sur les valeurs par défaut du serveur d'application conduit souvent à propager des messages d'erreur détaillés, pouvant contenir des attaques XSS par réflexion (« *The resource `<script>alert('xss')</script>.pdf` could not be found* »).

Journaliser des paramètres HTTP arbitraires

Un problème courant, transverse dans la sécurité d'une application, concerne la journalisation et la surveillance des actions de l'utilisateur. Bien qu'un *Front Controller* soit situé idéalement pour journaliser des requêtes entrantes, évitez de journaliser des requêtes HTTP complètes. Les requêtes HTTP contiennent des paramètres fournis par l'utilisateur, qui incluent souvent des informations confidentielles telles que les mots de passe ou les numéros de carte de crédit et des informations d'identification personnelle telle qu'une adresse. Journaliser des données confidentielles ou des informations d'identification personnelle contrevient aux règlements sur la sécurité et la vie privée.

A utiliser pour implémenter

Mapping des ressources logiques

La stratégie *Logical Resource Mapping* oblige les développeurs à définir explicitement les ressources disponibles pour le téléchargement et évite ainsi l'attaque par traversée de répertoire.

Gestion de session

La gestion de session est habituellement un des premiers contrôles de sécurité que l'application effectue sur une requête. En plus des contrôles de session gérés par le container tels que la durée d'inactivité ou l'invalidation, certaines applications implémentent des contrôles comme une durée maximale de session, la rotation de session et le lien avec l'adresse IP, par du code propriétaire. Utilisez le *Front Controller* pour effectuer les contrôles supplémentaires de gestion de session, avant l'exécution de chaque requête.

Invalidez le jeton de session courante et affectez un nouveau jeton de session après l'authentification est une défense courante contre les attaques par capture de session. Ce contrôle peut aussi être géré par le *Front Controller*.

Journalisation de l'audit

Comme les *Front Controllers* sont souvent conçus pour intercepter toutes les requêtes, ils sont les mieux placés pour réaliser la journalisation des actions de l'utilisateur à des fins d'audit.

Journalisez uniquement des informations comme :

- Nom de l'utilisateur pour les requêtes authentifiées
- Date et heure de la requête



- Ressource demandée
- Type de réponse : succès, erreur etc.

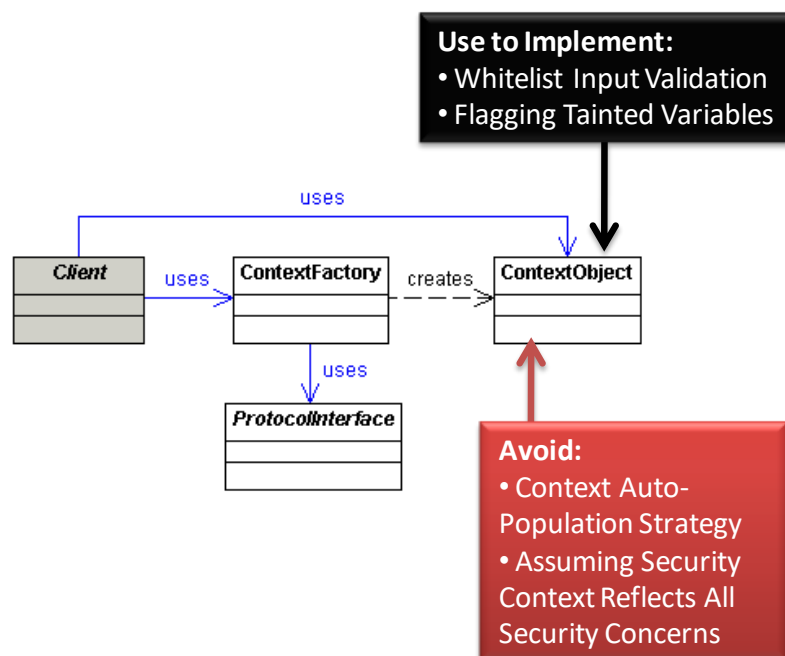
Ne journalisez jamais des données confidentielles ou personnelles.



CONTEXT OBJECT

In a multi-tiered applications, one tier may retrieve data from an interface using a specific protocol and then pass this data to another tier to be used for processing or as input into decision logic. In order to reduce the dependency of the inner tiers on any specific protocol, the protocol-specific details of the data can be removed and the data populated into a `ContextObject` which can be shared between tiers. Examples of such data include HTTP parameters, application configuration values, or security data such as the user login information, defined by the [Request Context](#), [Configuration Context](#), and [Security Context](#) strategies, respectively. By removing the protocol-specific details from the input, the *Context Object* pattern significantly reduces the effort required to adapt code to a change in the application's interfaces.

DIAGRAM



ANALYSIS

Avoid

Context Object Auto-Population Strategy

The [Context Object Auto-Population](#) strategy uses client-supplied parameters to populate the variables in a `Context` object. Rather than using a logical mapping to match parameters with `Context` variables, the [Context Object Auto-Population](#) strategy automatically matches `Context` variable names



with parameter names. In some cases, developers maintain two types of variables within a `Context` object: client-supplied and server supplied. An e-commerce application, for example, might have a `ShoppingCartContext` object with client-supplied product ID and quantity variables and a price variable derived from a server-side database query. If a client supplies a request such as “`http://siteurl/controller?command=purchase&prodID=43quantity=2`” then the [Context Object Auto-Population](#) strategy will automatically set the `ShoppingCartContext.prodId=43` and `ShoppingCartContext.quantity=2`. What if the user appends “`&price=0.01`” to the original query? The strategy automatically sets the `ShoppingCarContext.price=0.01` even though the price value should not be client controlled. Ryan Berg and Dinis Cruz and of Ounce labs documented this as a vulnerability in the Spring Model View Controller (MVC) framework⁹.

Avoid using the [Context Object Auto-Population](#) strategy wherever possible. If you must use this strategy, ensure that the user is actually allowed to supply the variables to the context object by performing explicit authorization checks.

Assuming Security Context Reflects All Security Concerns

The [Security Context](#) strategy should more precisely be called an Access Control Context strategy. Developers often assume that security is comprised entirely of authentication, authorization and encryption. This line of thinking often leads developers to believe that using the Secure Socket Layer (SSL) with user authentication and authorization is sufficient for creating a secure web application.

Also remember that fine-grained authorization decisions may be made further downstream in the application architecture, such as at the [Business Delegate](#). Consider propagating roles, permissions, and other relevant authorization information via the `Context` object.

Use to Implement

Whitelist Input Validation

The [Request Context Validation](#) strategy uses the `RequestContext` object to perform validation on client-supplied values. The Core J2EE Patterns book provides examples for form and business logic level validation, such as verifying the correct number of digits in a credit card. Use the same mechanism to perform security input validation with regular expressions. Unlike [Intercepting Filters](#), `RequestContexts` encapsulate enough context data to perform whitelist validation. Many developers employ this strategy in Apache Struts applications by opting to use the Apache Commons Validator^{vi} plugin for security input validation.

In several real-world implementations of [Request Context Validation](#), the `RequestContext` only encapsulates HTTP parameters. Remember that malicious user-supplied input can come from a variety of other sources: cookies, URI paths, and other HTTP headers. If you do use the [Request Context Validation](#) strategy for security input validation then provide mechanisms for security input validation on other forms of input. For example, use an [Intercepting Filter](#) to validate cookie data.



Flagging Tainted Variables

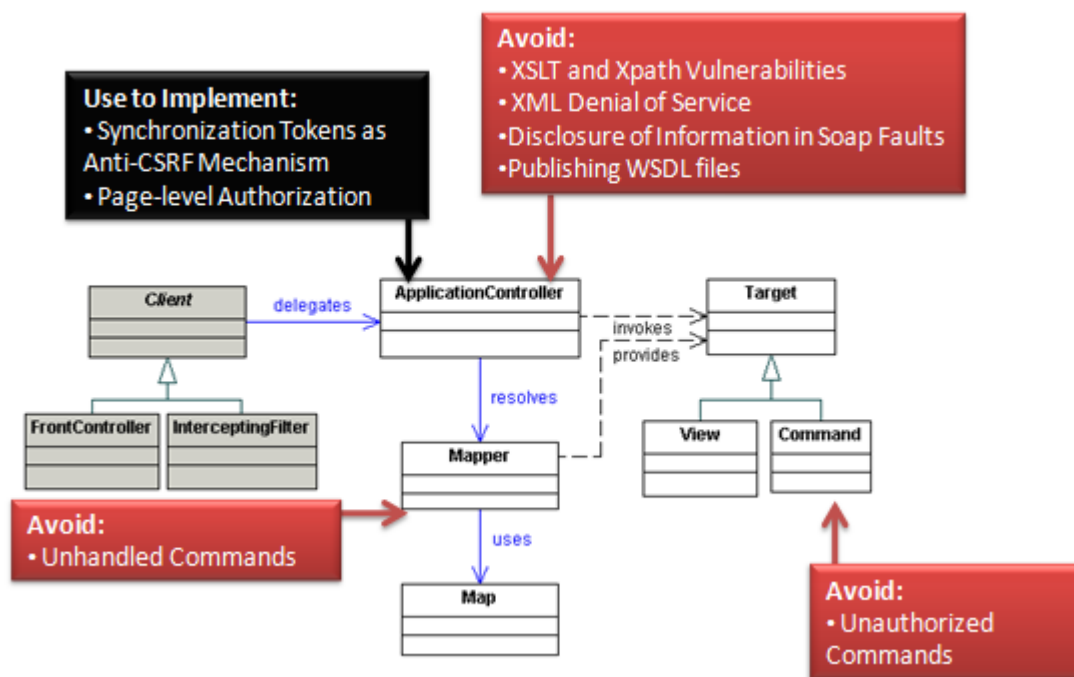
Conceptually, `Context` objects form the last layer where applications can differentiate between untrusted user-supplied data and trusted system-supplied data. For instance, a shopping cart `Context` object might contain a user-supplied shipping address and a database-supplied product name. Generally speaking, objects logically downstream from the `Context` object cannot distinguish user-supplied data from system-supplied data. If you encode data at sink points and you want to minimize performance impact by encoding only user-supplied data (as opposed to system generated data), then consider adding an “isTainted” flag for each variable in the `Context` class. Set “isTainted” to true if the variable is user supplied or derived from another user supplied value. Set “isTainted” to false if the variable is computer generated and can be trusted. Store the instance variable and “isTainted” booleans as key/value pairs in a collection with efficient lookups (such as a `WeakHashMap`). Downstream in the application, simply check if a variable is tainted (originates from user-supplied input) prior to deciding to encode it at sink points. For instance, you might HTML, JavaScript, or Cascading Stylesheet (CSS) encode all tainted data that you print to stream in a `Servlet` while leaving untainted data as is.



APPLICATION CONTROLLER

Alors que le pattern *Front Controller* centralise la logique commune à toutes les requêtes entrantes, la logique conditionnelle chargée de mapper chaque requête avec une commande et une vue doit être séparée. Le *FrontController* réalise tout le traitement générique des requêtes et délègue la logique conditionnelle à un *ApplicationController*. Celui-ci peut alors décider, en se basant sur la requête, quelle commande devrait servir la requête et vers quelle vue il faut naviguer. L'objet *ApplicationController* peut être étendu pour inclure de nouveaux *use case*, à travers une *map* contenant les références à la commande et à la vue, pour chaque transaction. Le pattern *Application Controller* est central dans le *framework MVC Struts*.

DIAGRAMME



ANALYSE

A éviter

Les commandes non autorisées

Dans la stratégie *Command Handler*, les utilisateurs fournissent un objet *Command* que l'objet *ApplicationController* va ensuite utiliser en exécutant une action. Les développeurs qui se



reposent sur les contrôles côté client ou au niveau des pages Web, oublient souvent de tester si l'utilisateur a réellement le droit d'appeler un objet `Command` donné.

Commandes inexistantes

Créez une page de réponse pour les commandes inexistantes. S'appuyer sur les valeurs par défaut du serveur d'application conduit souvent à propager des messages d'erreur détaillés, pouvant contenir des attaques XSS par réflexion (« *The resource `<script>alert('xss')</script>.pdf` could not be found* »).

Vulnérabilités XSLT et XPath

La stratégie **Transform Handler** utilise XSLT (*XML Stylesheet Language Transform*) pour générer des vues. Evitez les vulnérabilités XSLT et XPath, en validant strictement toutes les entrées par une liste blanche ou un encodage XML sur toutes les données fournies par l'utilisateur dans la génération des vues.

Déni de service XML

Si vous utilisez un *Application Controller* avec des messages XML, souvenez-vous que les attaquants peuvent tenter des attaques par Déni de Service (DOS) sur les parseurs et validateurs XML. Assurez-vous que le serveur web, le serveur d'application ou un *Intercepting Filter* effectue une vérification (*sanity check*) de la *taille* du message XML **avant** le *parsing* ou la validation, pour éviter les conditions du DOS.

Divulgaration d'information dans les erreurs SOAP

Une des vulnérabilités de divulgation d'information les plus répandues dans les web services survient quand des messages d'erreur renferment la *full stack trace* et/ou d'autres détails internes. Les *stack traces* sont souvent incluses par défaut dans les erreurs SOAP. Retirez cette option et retournez aux clients, des messages d'erreur génériques.

Publier des fichiers WSDL

Les fichiers web WSDL (*Web Services Description Language*) fournissent des détails sur la manière d'accéder aux services web et ils sont très utiles pour les attaquants. De nombreux frameworks SOAP publient le WSDL par défaut (<http://url/path?WSDL>).

Retirez cette option.

A utiliser pour implémenter

Synchronisation du jeton comme mécanisme anti-CSRF

Les jetons de synchronisation sont des nombres aléatoires conçus pour détecter les requêtes vers des pages web dupliquées. Utilisez des jetons fortement aléatoires pour protéger le mécanisme anti-CSRF (*Cross Site Request Forgery*). Souvenez-vous toutefois que les jetons CSRF peuvent être mis en échec, si



un attaquant parvient à lancer une attaque XSS (*Cross Site Scripting*) sur l'application, pour parser par programme et lire le jeton de synchronisation.

Autorisation au niveau de la page

Si vous n'utilisez pas encore un *Intercepting Filter*, utilisez l'objet *Application Controller* pour examiner les requêtes du client et s'assurer que seuls les utilisateurs autorisés peuvent accéder à une page donnée. Centraliser les contrôles d'autorisation enlève la charge d'inclure explicitement des autorisations par page, plus profondément dans l'application.

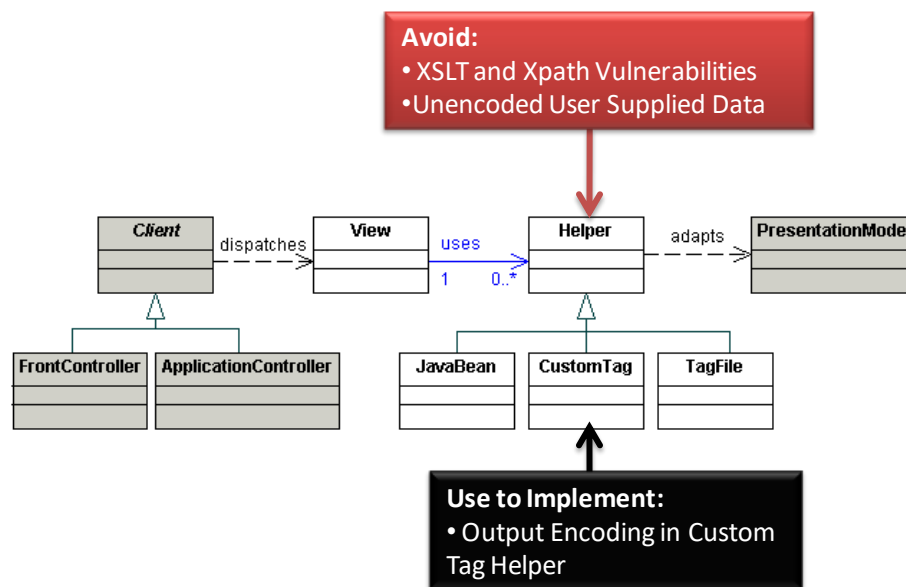
Souvenez-vous que l'autorisation au niveau de la page n'est qu'un seul composant du schéma complet d'autorisation. Gérez les autorisations au niveau de la commande si vous utilisez des objets *Command*, au niveau des paramètres des requêtes HTTP, et au niveau de la logique métier par des patterns comme *Business Delegate* ou *Session Façade*. Pensez à propager les informations sur le contrôle d'accès de l'utilisateur, comme le rôle de l'utilisateur, dans les autres couches de conception.



VIEW HELPER

View processing occurs with each request and typically consists of two major activities: processing and preparation of the data required by the view, and creating a view based on a template using data prepared during the first activity. These two components, often termed as `Model` and `View`, can be separated by using the *View Helper* pattern. Using this pattern, a view only contains the formatting logic either using the *Template-Based View* strategy such as a JSP or *Controller-Based View Strategy* using Servlets and XSLT transformations. The `View` then makes use of `Helper` objects that both retrieve data from the `PresentationModel` and encapsulate processing logic to format data for the `View`. *JavaBean Helper* and *Custom Tag Helper* are two popular strategies which use different data types (JavaBeans and custom view tags respectively) for encapsulating the model components.

DIAGRAM



ANALYSIS

Avoid

XSLT and XPath Vulnerabilities

Some developers elect to use Xml Stylesheet Language Transforms (XSLTs) within their `Helper` objects. Avoid XSLT^{vii} and related XPath^{viii} vulnerabilities by performing strict whitelist input validation or XML encoding on any user-supplied data used in view generation.

Unencoded User Supplied Data



Many JEE developers use standard and third party tag libraries extensively. Libraries such as Java Server pages Tag Libraries (JSTL)^{ix} and Java Server Faces (JSF)^x simplify development by encapsulating view building logic and hiding difficult-to-maintain scriptlet code. Some tags automatically perform HTML encoding on common special characters. The “c:out” and “\${fn:escapeXml(variable)}” Expression Language (EL) tags in JSTL automatically HTML encode potentially dangerous less-than, greater-than, ampersand, and apostrophe characters. Encoding a small subset of potentially malicious characters significantly reduces the risk of common Cross Site Scripting (XSS) attacks in web applications but may still leave other less-common malicious characters such as percentage signs unencoded. Many other tags do not perform any HTML encoding. Most do not perform any encoding for other sink types, such as JavaScript or Cascading Style Sheets (CSS).

Assume tag libraries do not perform output encoding unless you have specific evidence to the contrary. Wherever possible, wrap existing tag libraries with custom tags that perform output encoding. If you cannot use custom tags then manually encode user supplied data prior to embedding it in a tag.

Use to Implement

Output Encoding in Custom Tag Helper

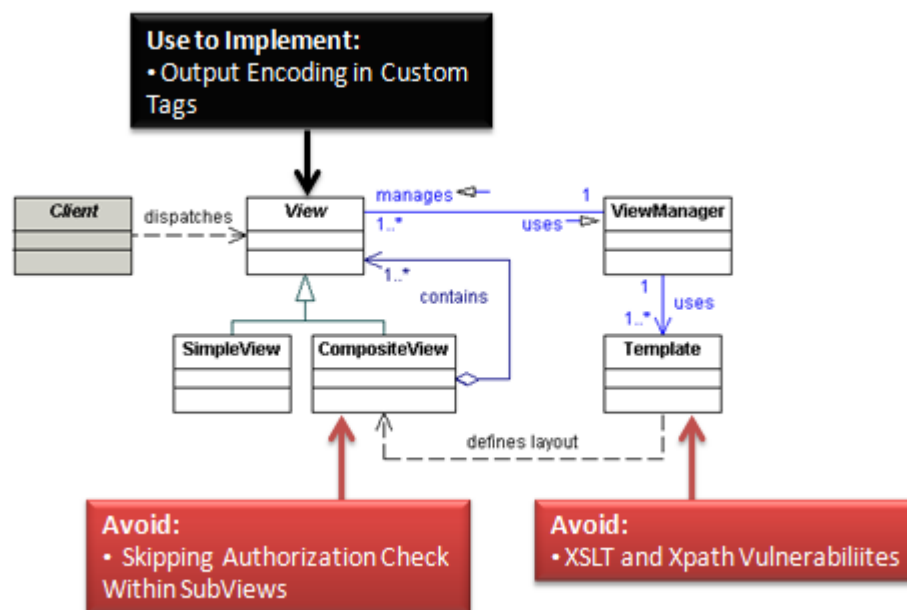
The [Custom Tag Helper](#) strategy uses custom tags to create views. Wherever possible embed output encoding in custom tags to automatically protect against Cross Site Scripting (XSS) attacks.



COMPOSITE VIEW

Applications often contain common View components, both from layout and content perspectives, across multiple Views in an application. These common components can be modularized by using the *Composite View* pattern which allows for a View to be built from modular, atomic components. Examples of modular components which can be reused are page headers, footers, and common tables. A CompositeView makes use of a ViewManager to assemble multiple views. A CompositeView can be assembled by JavaBeans, standard JSP tags such as `<jsp:include>`, or custom tags, using the *JavaBean View Management*, *Standard Tag View Management*, or *Custom Tag View Management* strategies respectively.

DIAGRAM



ANALYSIS

Avoid

XSLT and XPath Vulnerabilities

The *Transformer View Management* strategy uses XML Stylesheet Language Transforms (XSLTs). Avoid XSLT^{xi} and related XPath^{xii} vulnerabilities by performing strict whitelist input validation or XML encoding on any user-supplied data used in view generation.



Skipping Authorization Check Within SubViews

One of the most common web applications vulnerabilities is weak functional authorization. Developers sometimes use user role data to dynamically generate views. In the Core J2EE Pattern books the authors refer to these dynamic views as “Role-based content”. Role-based content prevents unauthorized users from *viewing* content but does not prevent unauthorized users from *invoking* Servlets and Java Server Pages (JSPs). For example, imagine a JEE accounting application with two JSPs – `accounting_functions.jsp` which is the main accounting page and `gl.jsp` representing the general ledger. In order to restrict access to the general ledger, developers include the following content in `accounts.jsp`:

```
<region: render template='portal.jsp'>
    <region:put section='general_ledger' content='gl.jsp'
        role='accountant' />
</region:render>
```

The code snippet above restricts the content in `gl.jsp` to users in the accountant role. Remember, however, that a user can simply access `gl.jsp` directly – a flaw that can be even more devastating if invoking `gl.jsp` with parameters causes a transaction to run such as posting a debit or credit.

Use to Implement

Output Encoding in Custom Tags

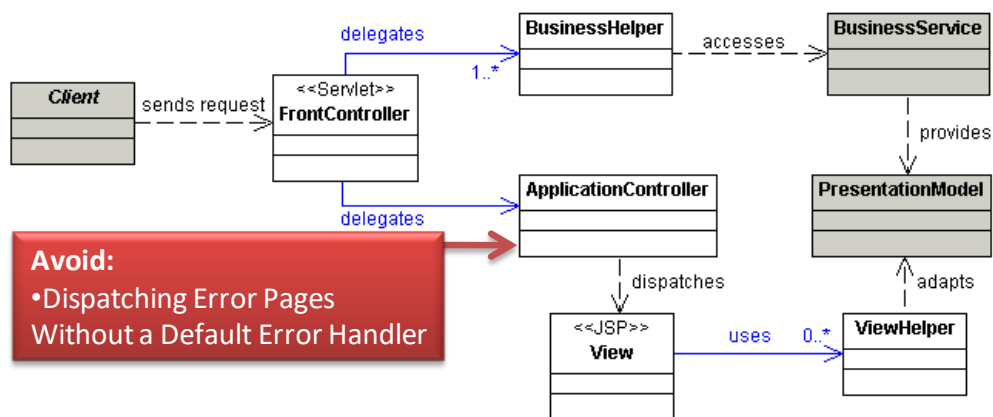
The [Custom Tag View Management](#) strategy uses custom tags to create views. Wherever possible embed output encoding in custom tags to automatically protect against Cross Site Scripting (XSS) attacks.



SERVICE TO WORKER

Applications commonly dispatch a `View` based exclusively on the results of request processing. In this the *Service to Worker* pattern, the `Controller` (either `FrontController` or `ApplicationController`) performs business logic and, based on the result of that logic, creates a `View` and invokes its logic. *Service to Worker* is different from *Dispatcher View* because in the latter the `View` logic is invoked **before** the business logic is invoked.

DIAGRAM



ANALYSIS

Avoid

Dispatching Error Pages Without a Default Error Handler

In *Service to Worker*, an `ApplicationController` manages the flow of web pages. When an application encounters an error, the `ApplicationController` typically determines which error page to dispatch. The Core J2EE Patterns book uses the following line of code to determine which error page to forward the user to:

```
next= ApplicationResources.getInstance().getErrorPage(e);
```

Many developers use the approach of mapping exceptions to particular error pages, but do not account for a default error page. As applications evolve, the job of creating a friendly error page for every exception type becomes increasingly difficult. In many applications the `ApplicationController` will simply dump a stack trace or even redisplay the client's request back to them. Avoid both scenarios by providing a default generic error page. In Java EE you can implement default error handling through `web.xml` configuration.

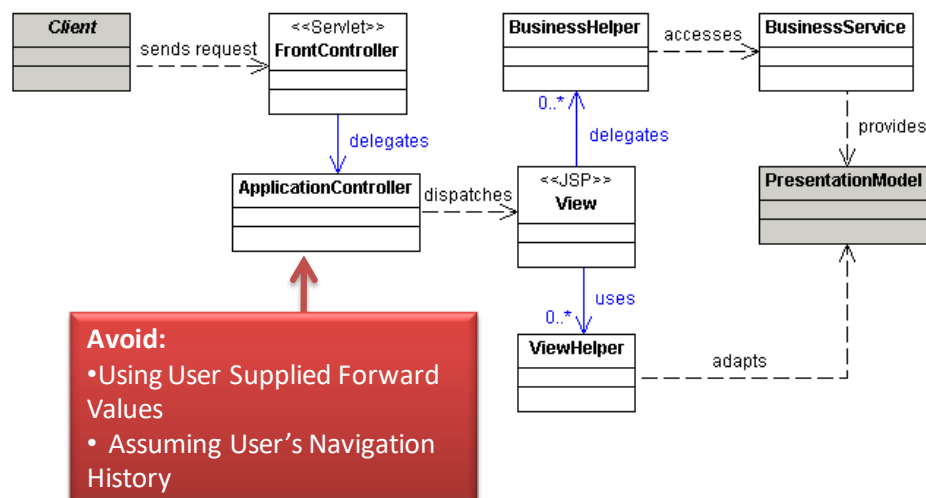




DISPATCHER VIEW

In instances where there is a limited amount or no business processing required prior to generating a response, control can be passed to the `View` directly from the `ApplicationController`. The `ApplicationController` performs general control logic and is only responsible for mapping the request to a corresponding `View`. **Dispatcher View** pattern is typically used in situations where the response is entirely static or the response is generated from data which has been generated from a previous request. Using this pattern, view management is often so trivial (i.e. mapping of an alias to a resource) that no application-level logic is required and it is handled by the container.

DIAGRAM



ANALYSIS

Avoid

Using User Supplied Forward Values

The `ApplicationController` determines which view to dispatch to a user. In the Core J2EE Patterns book, the authors provide the following sample of code to determine the next page to dispatch to a user:

```
next = request.getParameter("nextview");
```

Depending on how you forward users to the next page, an attacker may be able to:



- Gain unauthorized access to pages if the forwarding mechanism bypasses authorization checks
- Forward unsuspecting users to a malicious site. Attackers can take advantage of the forwarding feature to craft more convincing phishing emails with links such as `http://siteurl/page?nextview=http://malicioussite/attack.html`. Since the URL originates from a trusted domain – “http://siteurl” in the example – it is less likely to be caught by phishing filters or careful users
- Perform Cross Site Scripting (XSS) in browsers that accept JavaScript URLs. For example, an attacker might send a phishing email with the following URL:
http://siteurl/page?nextview=javascript:do_something_malicious
- Perform HTTP Response Splitting if the application uses an HTTP redirect to forward the user to the next page. The user supplies the literal value of the destination URL. Because the application server uses that URL in the header of the HTTP redirect response, a malicious user can inject carriage return and line feeds into the response using hex encoded %0A and %0D

Do not rely on literal user-supplied values to determine the next page. Instead, use a logical mapping of numbers to actual pages. For example, in `http://siteurl/page?nextview=1` the “1” maps to “edituser.jsp” on the server.

Assuming User’s Navigation History

The Core J2EE Patterns book discusses an example where the *Dispatcher View* pattern may be used. In the example, the application places an intermediate model in a temporary store and a later request uses that model to generate a response. When writing code for the second view, some developers assume that the user has already navigated to the first view thereby instantiating and populating the intermediate model. An attacker may take advantage of this assumption by directly navigating to the second view before the first. Similarly, if the first view automatically dispatches the second view (through an HTTP or JavaScript redirect), an attacker may drop the second request, resulting in an unexpected state. Always use server-side checks such as session variables to verify that the user has followed the intended navigation path.



PATTERNS DE LA COUCHE METIER

Sécurité des couches intermédiaire et intégration (*Middle and Integration Tier Security*)

La majorité des vulnérabilités les plus répandues (Top 10 de l'OWASP) surviennent dans la couche de présentation. Quand des attaques exploitent des vulnérabilités de la couche métier et intégration, elles viennent habituellement de la couche de présentation Web. Par exemple, une attaque par injection SQL transite par une requête HTTP envoyée à la page web, mais l'exploitation se fait dans la couche d'intégration.

Nous allons examiner les patterns de la couche métier et intégration, sous deux aspects :

1. Les attaques issues de la couche présentation, comme l'attaque Cross-Site Scripting (XSS) envoyée par un client web malveillant
2. Les attaques issues de la couche métier et intégration, telle qu'une requête non autorisée d'un service web à un appareil de la couche d'intégration.

Le deuxième aspect est suffisamment important pour mériter une attention particulière. Beaucoup d'organisations ne font pas attention aux attaques provenant de leur réseau interne. Malheureusement, le postulat affirmant qu'une organisation peut faire complètement confiance à ses membres, est faux. Dans le développement sécurisé, les développeurs soutiennent parfois qu'il est compliqué donc moins probable de lancer une attaque depuis la couche métier et intégration. Mais comme les organisations utilisent de plus en plus SOA (*Service Oriented Architectures*) dans la couche métier et intégration, avec des protocoles comme SOAP (*Simple Object Access Protocol*) et REST (*Representation State Transfer*), les outils de sécurité qui ciblent ces protocoles deviennent facilement disponibles.

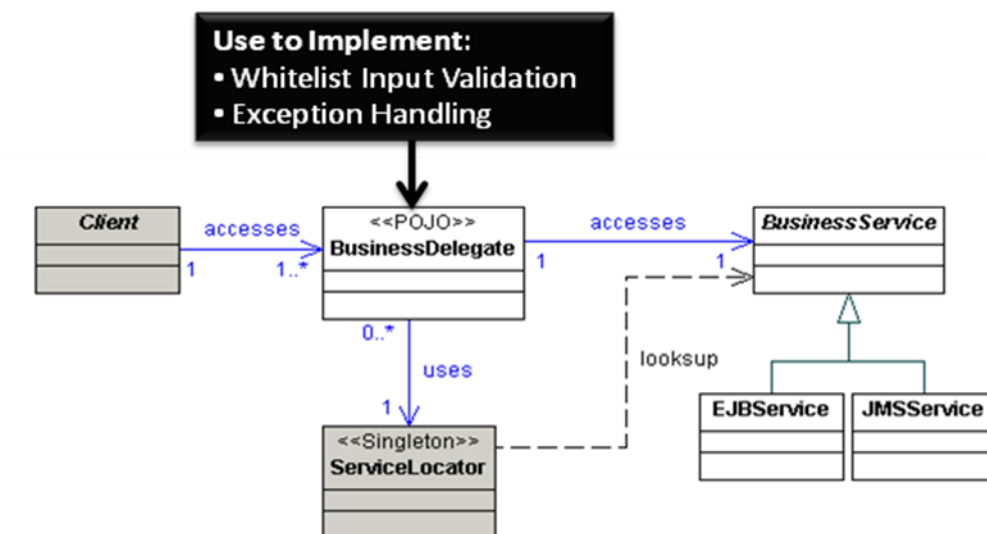
Même si vous n'avez jamais fait l'expérience d'un incident de sécurité interne, notez que les attaques internes augmentent, et que les systèmes que vous concevez aujourd'hui doivent rester en production pendant les années ou même les dizaines d'années à venir. Créez des applications sécurisées avec des protections intégrées contre les menaces internes.



BUSINESS DELEGATE

The *Business Delegate* serves as an abstraction of the business service classes of the business tier from the client tier. Implementing a business delegate effectively reduces the coupling between the client tier and business tier, and allows for greater flexibility and maintainability of the application code. The most significant benefit of this pattern is the capability to hide potentially sensitive implementation details of the business services from the calling client tier. Furthermore, a business delegate can effectively handle business tier exceptions (such as `java.rmi.Remote` exceptions) and translate them into more meaningful, application-level exceptions to be forwarded to the client.

DIAGRAM



ANALYSIS

Use to Implement

Whitelist input validation

The `DelegateProxyStrategy` uses `BusinessDelegate` objects as simple proxies to underlying services. Each `BusinessDelegate` is business context specific and is therefore a good place to implement whitelist security input validation. Remember, however, that `BusinessDelegate` validation only applies to input originating from the presentation tier. You need to duplicate input validation functionality for all other channels that access the same business tier, such as web services.

Exception Handling



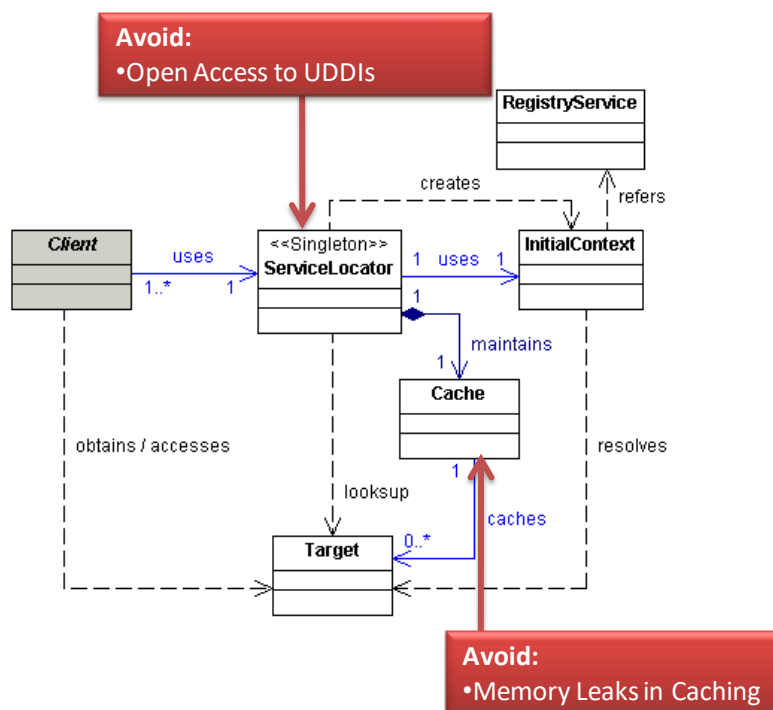
A property of sound exception management is the practice of throwing an exception that is meaningful to the target tier. For example, a `JMSEException` caught by a business tier object should not be propagated to the client tier. Instead, a custom, application-level, exception should be sent to the client tier. `BusinessDelegate` can effectively perform this translation, intercepting service-level exceptions from the business tier and throwing application-level exceptions to the client tier. This practice helps protect implementation-level details of the business services from calling clients. Note that exceptions can occur within and across many tiers, and restricting exception handling logic to the `BusinessDelegate` alone is insufficient.



SERVICE LOCATOR

JEE application modules from the client tier often need to access business or integration tier services, such as JMS components, EJB components, or data sources. These components are typically housed in a central registry. In order to communicate with this registry, the client uses the Java Naming and Directory Interface (JNDI) API to obtain an `InitialContext` object containing the desired object name. Unfortunately, this process can be repeated across several clients, increasing complexity and decreasing performance. The *Service Locator* pattern implements a `ServiceLocator` singleton class that encapsulates API lookups and lookup complexities, and provides an easy-to-use interface for the client tier. This pattern helps promote reuse of resource-intensive lookups, and decouples the client tier from the underlying lookup implementation details.

DIAGRAM



ANALYSIS

Avoid

Memory Leaks in Caching



Developers typically use caching to improve performance. If you are not careful about implementing caching, however, you can actually degrade performance over time.

Many developers use collections like `HashMap`s to maintain references to cached objects. Suppose you maintain a cache using a `HashMap` where the keys are `Strings` describing services in the `ServiceLocator` and the values are `Target` objects. After a period of inactivity you want to remove the `Target` reference to free memory. Simply running a method like `close()` on the `Target` object will not actually make the object eligible for garbage collection because the cache `HashMap` still maintains a pointer to the `Target` object. You must remember to remove references from the `HashMap` otherwise you will experience memory degradation overtime and possibly suffer from Denial of Service (DoS) conditions.

Use a `WeakHashMap` or another collection of `WeakReferences` or `SoftReferences` rather than traditional collections to maintain caches. A `WeakHashMap` maintains `WeakReferences`^{xiii} to key and value objects, meaning the cache will allow key and value objects to be eligible for garbage collection. Using `WeakReferences` allows the garbage collector to automatically remove objects from the cache when necessary, as long as the application does not maintain any strong references (i.e. regular pointers) to the object.

If you cannot use `WeakReferences` or `SoftReferences` then ensure you regularly remove objects from the cache.

Open Access to UDDI

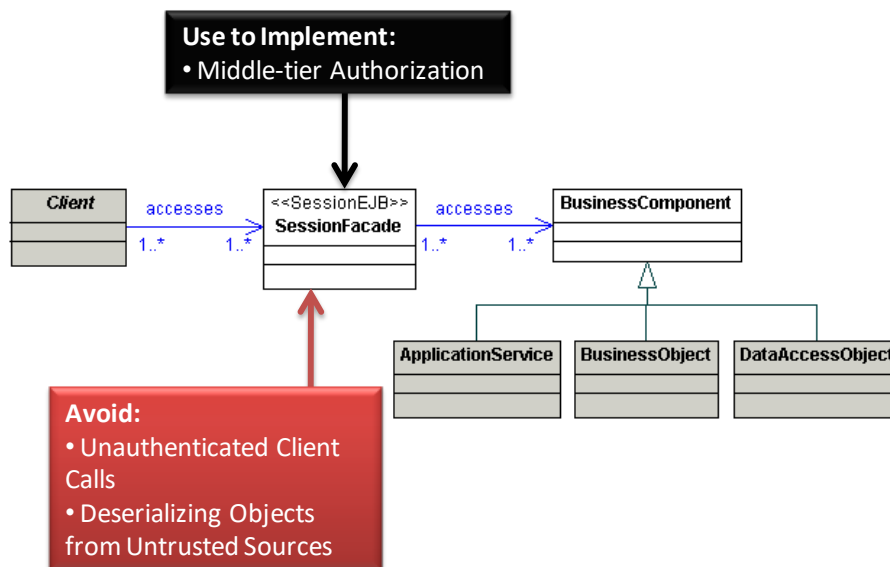
Many developers opt to use web services instead of more traditional middle tier technologies like Enterprise Java Beans (EJBs). In Service Oriented Architecture (SOAs), Universal Description, Discovery and Integration (UDDI), registries often play the role of Service Locator. UDDIs are extremely valuable to attackers since they house Web Service Definition Language (WSDL) files that provide blueprints of how and where to access web services. If you use UDDI, ensure that only authorized users can access registries. Consider using mutual certificate authentication with each UDDI request. Also ensure that an authenticated user is actually authorized to access specific parts of the registry.



SESSION FAÇADE

Les composants métier dans les applications JEE interagissent souvent avec de nombreux services, qui peuvent chacun inclure des processus complexes. Exposer directement ces composants à la couche client n'est pas souhaitable, car cela conduit à un couplage fort entre les couches client et métier, et peut conduire à dupliquer du code. Le pattern *Session Façade* est utilisé pour encapsuler les services de la couche métier, et sert d'interface pour le client. Les développeurs implémentent habituellement la classe `SessionFacade` comme un EJB session et expose uniquement les interfaces requises par le client, en cachant les interdépendances complexes entre les objets métier. Il ne faudrait placer que peu ou pas du tout de logique métier dans la `SessionFacade`.

DIAGRAM



ANALYSE

A éviter

Les appels d'un client non authentifié

Permettre un accès non authentifié à des EJB, des services web ou d'autres composants de la couche intermédiaire, laisse les applications vulnérables à des attaques internes. Dans certains cas, les développeurs configurent les serveurs applicatifs en autorisant un accès total aux sources de données, comme les bases de données. Il faut construire une authentification de système à système dans la *Session Façade*, ou utiliser des mécanismes gérés par le container, comme l'authentification mutuelle par certificat. Notez aussi que les sources de données peuvent être rendues accessibles par le serveur applicatif, pour quiconque les retrouve sur le réseau.



Désérialiser des objets depuis des sources non sûres

Les développeurs utilisent souvent le protocole RMI-IIOP (*Remote Method Invocation over Internet Inter-Orb Protocol*) pour communiquer entre la couche présentation et la couche métier. RMI utilise la sérialisation et la désérialisation Java pour transférer des objets à travers le réseau. Traitez la désérialisation comme une fonctionnalité dangereuse, car certaines machines virtuelles Java sont vulnérables au déni de service (DOS) quand les attaquants transmettent des objets sérialisés fabriqués spécialement pour cet objectif. Prenez soin de traiter les objets sérialisés envoyés par le client ou par un tiers, comme des entrées non sûres ; des utilisateurs malveillants peuvent modifier la version sérialisée d'un objet pour injecter des données malveillantes ou même pour fournir des objets faisant tourner du code malveillant.

Réduisez le risque d'une attaque par déni de service (DOS) en authentifiant les requêtes *avant* de désérialiser les données. Mettez à jour votre machine virtuelle Java, au même titre que les autres composants logiciels critiques comme les systèmes d'exploitation.

A utiliser pour implémenter

Autorisation de couche intermédiaire (*Middle Tier Authorization at Session Facade*)

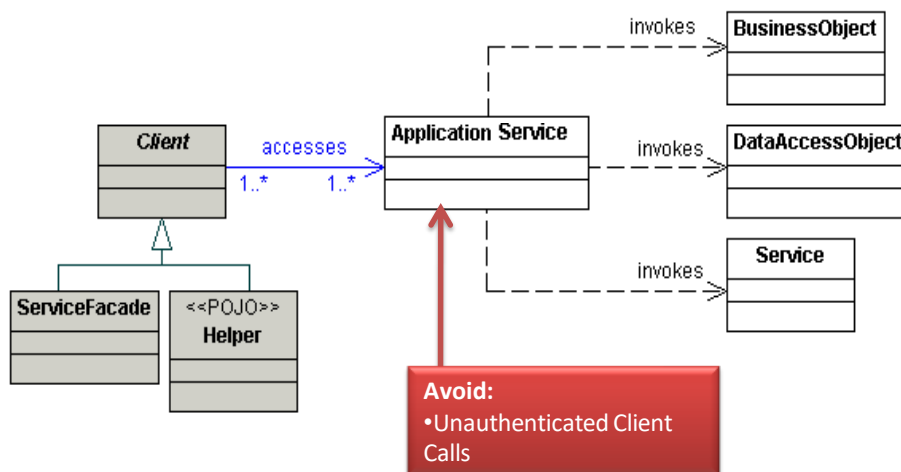
Les *Session Façades* sont les points d'entrée de la couche intermédiaire, ce qui veut dire qu'elles sont idéales pour gérer les autorisations de la couche intermédiaire. Assurez-vous que tous les clients ont des droits suffisants pour accéder à chacune des fonctions exposées à travers la *Session Façade*.



APPLICATION SERVICE

Bien que le pattern *Session Façade* permette de supprimer le couplage entre la couche client et la couche métier et encapsule les services de la couche métier, les objets *SessionFacade* ne devraient pas encapsuler la logique métier. Une application peut contenir plusieurs *BusinessObjects* (patterns suivants) qui remplissent des services différents, mais placer la logique métier dans l'objet *Session Façade* créerait un couplage entre eux. Le pattern *Application Service* fournit une couche de service uniforme pour le client. Une classe *ApplicationService* centralise la logique métier qui encapsule un service métier spécifique de l'application et réduit ainsi le couplage entre les *BusinessObjects*. L'objet *ApplicationService* peut implémenter la logique commune en agissant sur différents *BusinessObjects*, implémenter la logique métier d'un *Use Case*, appeler les méthodes d'un *BusinessObject*, ou les méthodes d'autres objets *ApplicationServices*. Un objet *ApplicationService* est le plus souvent implémenté comme un POJO (*Plain-Old Java Object*).

DIAGRAMME



ANALYSE

A éviter

Les appels d'un client non authentifié

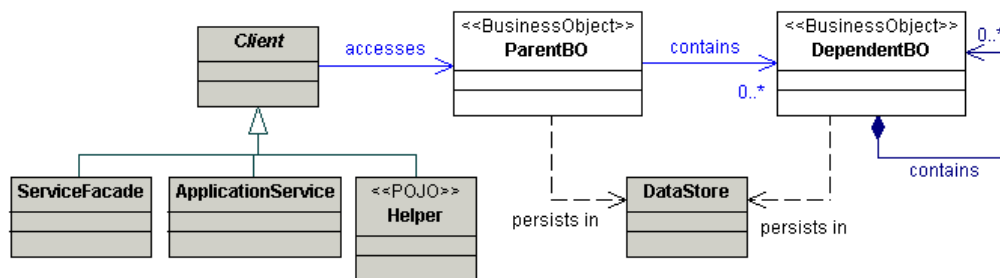
Permettre un accès non authentifié à des EJB, des services web ou d'autres composants de la couche intermédiaire, laissent les applications vulnérables à des attaques internes. Il faut construire une authentification dans l'objet *Application Service* ou utiliser des mécanismes gérés par le container, comme l'authentification mutuelle par certificat.



BUSINESS OBJECT

Idéalement, une application N Tiers se divise en couches de présentation, métier et données. Parmi les trois, la couche métier conduit souvent à une duplication de code et à un couplage fort avec la couche de données. Le pattern *Business Object* permet de séparer l'état et le comportement de la couche métier du reste de l'application, et à centraliser les deux. Les objets *BusinessObjects* encapsulent les données métier essentielles et les comportements métier attendus, tout en séparant la logique de persistance de la logique métier. En utilisant ce pattern, le client interagit directement avec un *BusinessObject* qui délègue son comportement à une ou plusieurs entités métiers, et gère sa propre logique de persistance en utilisant la stratégie de persistance disponible, comme des POJO ou des EJB entités.

DIAGRAMME



ANALYSE

Notes générales

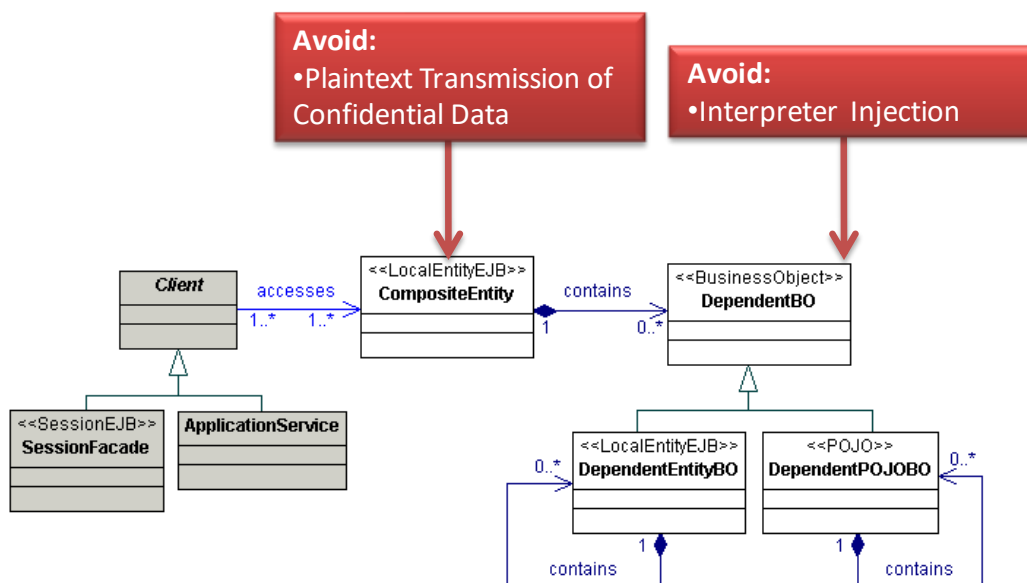
Dans la plupart des cas, les *Business Objects* n'implémentent aucune fonctionnalité de sécurité commune comme l'authentification ou l'autorisation, la validation des données et l'encodage, ou la gestion des sessions. Idéalement, un *Business Object* devrait simplement fournir des accesseurs et les fonctionnalités de base de la logique métier. Vous devez néanmoins les développer en respectant les pratiques habituelles du développement sécurisé, comme la gestion et la journalisation des erreurs.



COMPOSITE ENTITY

Business Objects separate business logic and business data. While Enterprise Java Beans (EJB) entity beans are one way to implement *Business Objects*, they come with complexity and network overhead. The *Composite Entity* pattern uses both local entity beans and Plain Old Java Objects (POJOs) to implement persistent *Business Objects*. The *Composite Entity* can aggregate a single *BusinessObject* and its related dependant *BusinessObjects* into coarse-grained entity beans. Using this pattern allows developers to leverage the EJB architecture, such as container-managed transactions, security, and persistence.

DIAGRAM



ANALYSIS

General Notes

Entity Bean Alternatives

With the rise of third party persistence frameworks such as Hibernate and IBATIS, fewer developers use entity beans. In general, application developers can take security advice for *Composite Entity* and apply it to persistence frameworks other than Entity Beans.

Avoid

Interpreter Injection



If you use bean managed persistence (BMP) with entity beans, then protect against injection attacks by using secure prepared statements, stored procedures, or appropriate encoding for non-database persistence mechanisms such as XML encoding for creating XML documents.

In most cases persistence frameworks like Hibernate automatically protect against SQL injection. Remember, however, that many persistence frameworks allow you to create custom queries through special functions like Hibernate Query Language (HQL). If you dynamically concatenate strings to create custom queries, then your application may be vulnerable to injection despite the fact that you use a persistence framework^{xiv}.

Plaintext Transmission of Confidential Data

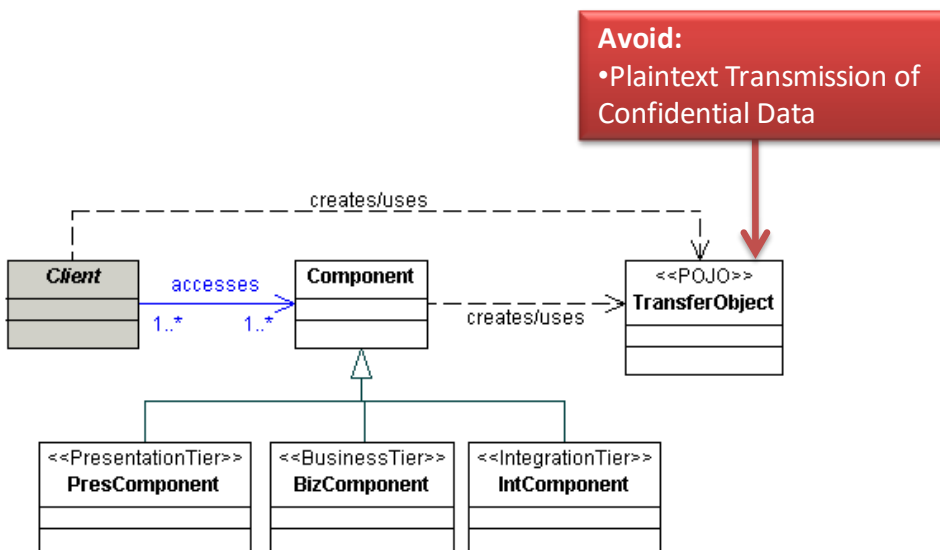
The [Composite Transfer Object](#) strategy creates a `TransferObject` to send to a remote client. Many organizations do not use encrypted communication channels within the internal network, so a malicious insider attacker might sniff confidential data within a `TransferObject` during transmission. Either do not serialize confidential variables or use an encrypted communication channel like Secure Socket Layer (SSL) during transmission.



TRANSFER OBJECT

Les patterns d'intégration comme *Session Façade* ou *Business Object* ont souvent besoin de retourner des données au client. Toutefois, le client peut être amené à appeler plusieurs méthodes *get* de ces objets, et quand ces patterns sont implantés comme des EJB distants, cela génère un trafic réseau important à chaque appel. Le pattern *Transfer Object* est prévu pour optimiser le transfert de données vers la couche client. Un *TransferObject* encapsule les données élémentaires dans une seule structure, qu'elle retourne au client appelant. Le pattern *Transfer Object* sert à réduire le trafic réseau, à transférer des grosses quantités de données avec moins d'appels distants et de faciliter la réutilisation du code.

DIAGRAM



ANALYSE

A éviter

La transmission en clair de données confidentielles

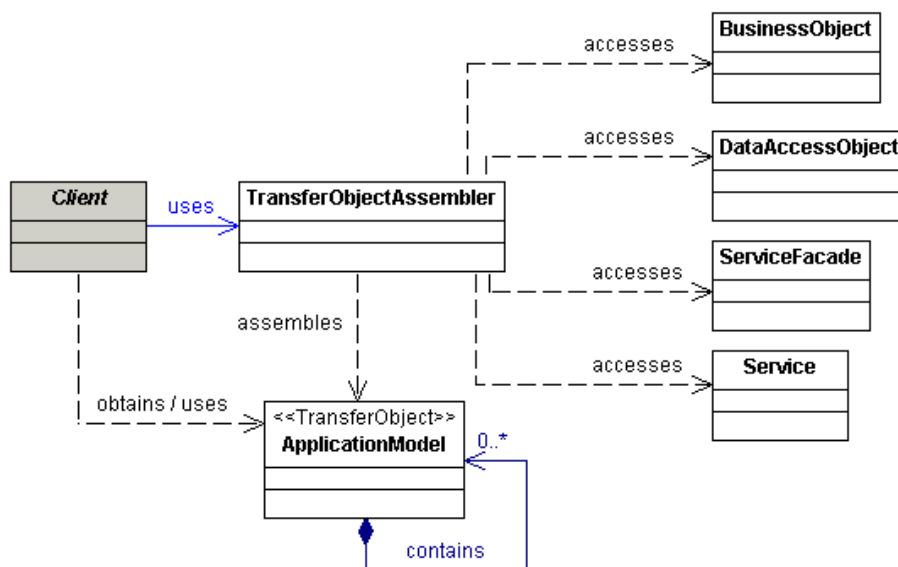
De nombreuses organisations ne chiffrent pas leurs canaux de communication dans leur réseau interne, et un utilisateur interne malveillant pourrait donc capturer des données confidentielles sur le réseau, pendant la transmission d'un objet *TransferObject*. Donc, il faut : soit ne pas sérialiser des données confidentielles, soit utiliser un canal chiffré comme SSL (*Secure Socket Layer*) pendant la transmission.



TRANSFER OBJECT ASSEMBLER

Les clients ont souvent besoin d'accéder à des données métier à travers différents objets `BusinessObjects`, `TransferObjects`, ou des objets `ApplicationServices` pour réaliser les traitements. Accéder directement à différents composants depuis le client, conduirait à un couplage fort entre les différents tiers et à dupliquer le code. Le pattern *Transfer Object Assembler* fournit un moyen efficace pour assembler plusieurs `TransferObjects` à partir de différents composants métier et de renvoyer l'information au client. Considérez le *Transfer Object Assembler* comme une fabrique (*factory*) pour créer les objets `TransferObject`. Le client appelle le `TransferObjectAssembler`, qui récupère et traite différents `TransferObjects` issus de la couche métier, et construit un `TransferObject` composé, nommé `ApplicationModel` qu'il retourne au client. Notez que le client utilise l'objet `ApplicationModel` en lecture seule.

DIAGRAMME



ANALYSE

Appels d'un client non authentifié

Autoriser des accès non authentifiés à un `TransferObjectAssemblers` ou à d'autres composants de la couche intermédiaire laisse les applications vulnérables aux attaques venant de l'intérieur. Si vous exposez directement votre `TransferObjectAssemblers` aux clients, alors construisez une



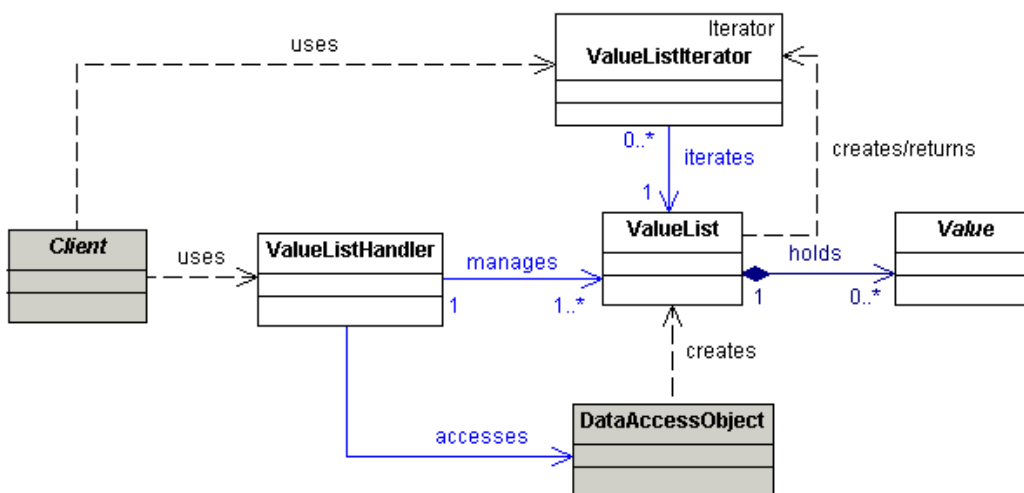
authentification de système à système ou utilisez les mécanismes gérés par le container comme l'authentification mutuelle par certificat.



VALUE LIST HANDLER

Searching is a common operation in JEE applications. A search typically is initiated by the client tier and execution by the business tier. More concretely, this can involve several invocations of entity bean finder methods or `DataAccessObjects` by the client, particularly if the result set of the search is large. This introduces network overhead. The *Value List Handler* pattern affords efficient searching and result set caching, which can allow the client to traverse and select desired objects from the result set. A `ValueListHandler` object executes the search and obtains the results in a `ValueList`. The `ValueList` is normally created and manipulated by the `ValueListHandler` through a `DataAccessObject`. The `ValueList` is returned to the client, and the client can iterate through the list contents using a `ValueListIterator`.

DIAGRAM



ANALYSIS

General Notes

In most cases, *Value List Handlers* do not implement common security features such as authentication and authorization, data validation and encoding, or session management. You still need to develop with standard secure coding practices such as error handling and logging^{xv}.

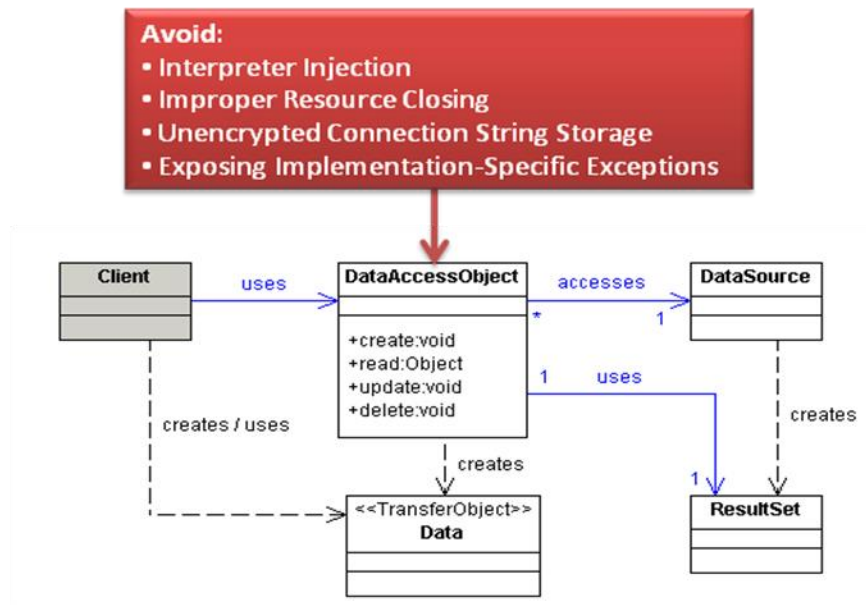


PATTERNS DE LA COUCHE INTEGRATION

DATA ACCESS OBJECT

Presque toutes les applications Java EE ont besoin d'accéder à des données d'entreprise ou des données métier, depuis un magasin de données persistantes. Toutefois, les données peuvent être situées dans différents types de stockage, bases de données, *mainframe* ou vieux systèmes. Mélanger la logique applicative avec la logique de persistance introduit un couplage fort et transforme la maintenance de l'application en un cauchemar. Le pattern *Data Access Object* (DAO) permet d'encapsuler tous les accès au magasin de données persistantes et de gérer les connexions vers la couche de données, en cachant au client les détails d'implémentation de l'API de persistance. Un objet DAO implémente les opérations *create*, *find*, *update*, et *delete*.

DIAGRAMME



ANALYSE

A éviter

L'injection (*Interpreter Injection*)

Les objets *DAO* interagissent avec des interpréteurs comme SQL ou LDAP (*Lightweight Directory Access Protocol*). Ils sont donc vulnérables aux attaques par injection comme l'injection SQL. Eviter les attaques par injection SQL, en utilisant des requêtes paramétrées bien construites ou des procédures stockées.



Pour d'autres interpréteurs, comme les magasins LDAP ou XML, utilisez le bon encodage pour échapper les caractères potentiellement malveillants. Prenez soin d'utiliser une approche par liste blanche plutôt que par liste noire, pour encoder.

Il se peut que pour certains interpréteurs vous ne trouviez pas de méthode d'encodage, ce qui laisserait votre code très vulnérable aux injections pour ces interpréteurs. Dans ce cas, utilisez une validation stricte des entrées par liste blanche.

Mauvaise fermeture d'une ressource

Les développeurs oublient souvent de fermer proprement les ressources, telles que les connexions aux bases de données. Fermez toujours les ressources dans un bloc *finally*, vérifiez que le pointeur n'est pas nul avant de fermer un objet, et gérez par un *catch*, toute exception qui pourrait survenir pendant la fermeture de la ressource, dans le bloc *finally*.

Stockage d'une chaîne de connexion non chiffrée

Pour communiquer avec une base de données ou un autre serveur de *backend*, beaucoup d'applications utilisent une chaîne de connexion, qui inclut un nom d'utilisateur et un mot de passe. Les développeurs stockent souvent cette chaîne de connexion non chiffrée dans des fichiers de configuration du serveur comme *server.xml*. Des utilisateurs internes malveillants ou des attaquants externes peuvent utiliser cette chaîne de connexion en clair, pour accéder de façon non autorisée à la base de données.

Chiffrez les informations de sécurité de la base de données et de tout autre serveur, avant de les stocker. Les moyens de chiffrement fournis par les serveurs applicatifs ne sont pas parfaits, mais stocker un mot de passe en clair reste cependant la solution la moins sécurisée.

Exposer des exceptions dépendant de l'implémentation

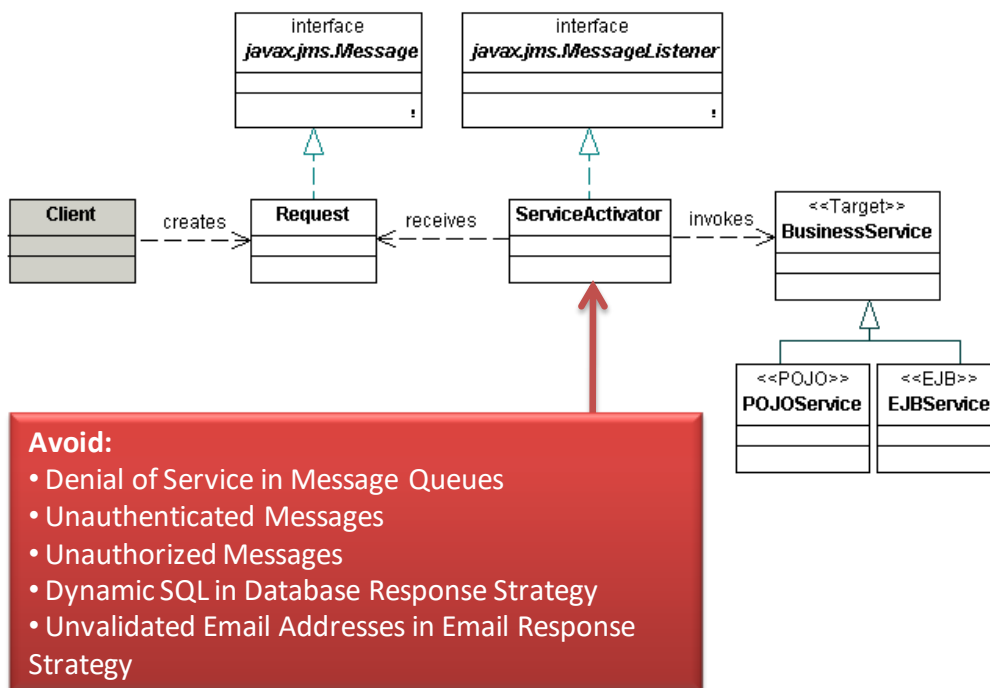
Un objet DAO crée une connexion directe à la couche de persistance, lui envoie des requêtes, et retourne les résultats au client. Un tel accès exige de gérer et de traduire les exceptions qui proviennent de la couche de données. Assurez-vous que les exceptions qui sont levées par les packages spécifiques à l'implémentation, comme `java.sql.*` ou `javax.sql.*` sont prises en compte et mémorisées dans l'objet DAO, et ne sont pas simplement propagées vers le client. Une stratégie répandue est de mémoriser l'exception dans l'objet DAO et d'envoyer au client une exception créée spécifiquement dans ce but, comme `DAOException`.



SERVICE ACTIVATOR

In JEE applications, certain business services can involve complex processes that may consume considerable time and resources. In such cases, developers often prefer to invoke these services asynchronously. The *Service Activator* pattern allows for the reception of asynchronous requests from a client and invokes multiple business services. A *ServiceActivator* class is typically implemented as a JMS listener that receives and parses messages from the client, identifies and invokes the correct business service to process the request, and informs the client of the outcome.

DIAGRAM



ANALYSIS

Avoid

Denial of Service in Message Queues

Attackers may fill up message queues to launch a Denial of Service (DOS) attacks. Perform a sanity check on the size of a message prior to accepting it into the message queue.

Unauthenticated Messages



Allowing unauthenticated access to `BusinessServices` or other middle tier components leaves applications susceptible to insider attacks. Build system-to-system authentication into the `ServiceActivator` itself or use container-managed mechanisms such as mutual certificate authentication.

Another approach is to add authentication credentials into each message, similar to WS-Security authentication in web services^{xvi}. Remember, however, that processor-intensive authentication functions may themselves leave applications vulnerable to DOS attacks.

Unauthorized Messages

`BusinessServices` often expose critical middle tier or backend functions. Include functional authorization checks to protect `BusinessServices` from authenticated user privilege escalation.

Developers sometimes skip middle tier authorization, relying instead on functional authorization checks in the presentation tier (e.g. application controller checks). A presentation-tier-only authorization approach might work with strong infrastructure-level access controls. Remember, however, that you must duplicate authorization checks on all channels that access the business service. Wherever possible use the `ServiceActivator` to enforce consistent access control for all messages.

Dynamic SQL in Database Response Strategy

The `Database Response` strategy stores the message response in a database that a client subsequently polls. As with other database interaction patterns, remember to use properly configure Prepared Statements or stored procedures to avoid SQL injection.

Unvalidated Email Addresses in Email Response Strategy

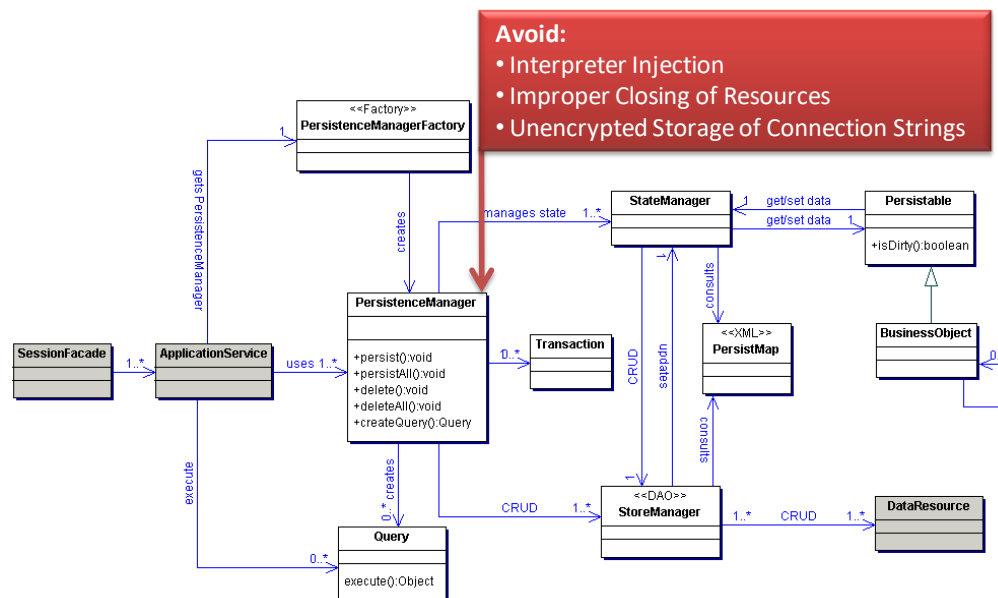
The `Email Response` strategy sends a message response though email. Malicious users who can influence the response may provide multiple email addresses and use the server as a gateway to send unauthorized spam messages. Use whitelist validation to ensure that users supply only one email address^{xvii}.



DOMAIN STORE

Patterns such as *Data Access Object* emphasize the value of separating persistence details from *BusinessObjects*. Some applications must also separate persistence details from the application object model. The *Domain Store* pattern allows for this separation, unlike container-managed persistence and bean-managed persistence strategies, which tie persistence code with the object model. The *Domain Store* pattern can be implemented using either a custom persistence framework or a persistence product, such as Java Data Objects.

DIAGRAM



ANALYSIS

Avoid

Interpreter Injection

Domain Stores interact with interpreters such as SQL-enabled databases or Lightweight Directory Access Protocol (LDAP) enabled repositories. DAOs are therefore susceptible to injection-style attacks such as SQL injection. Avoid SQL attacks by using properly constructed prepared statements or stored procedures. For other interpreters, such as LDAP or XML data stores, use appropriate encoding to escape potentially malicious characters such as LDAP encoding or XML encoding. Remember to use a whitelist approach for encoding rather than a blacklist approach.



You may not find encoding methods for some interpreters, leaving your code highly susceptible to interpreter injection. In these cases use strict whitelist input validation.

Properly Close Resources

Developers often forget to properly close resources such as database connections. Always close resources in a finally block, check for null pointers before closing on an object, and catch and handle any possible exception that may occur during resource closing *within* the finally block^{xviii}.

Connection String Storage

In order to communicate with a database or other backend server, most applications use a connection string that includes a user name and password. Most developers store this connection string unencrypted in server configuration files such as server.xml. Malicious insiders and external attackers who exploit path traversal vulnerabilities^{xix} may be able to use a plaintext connection string to gain unauthorized access to the database.

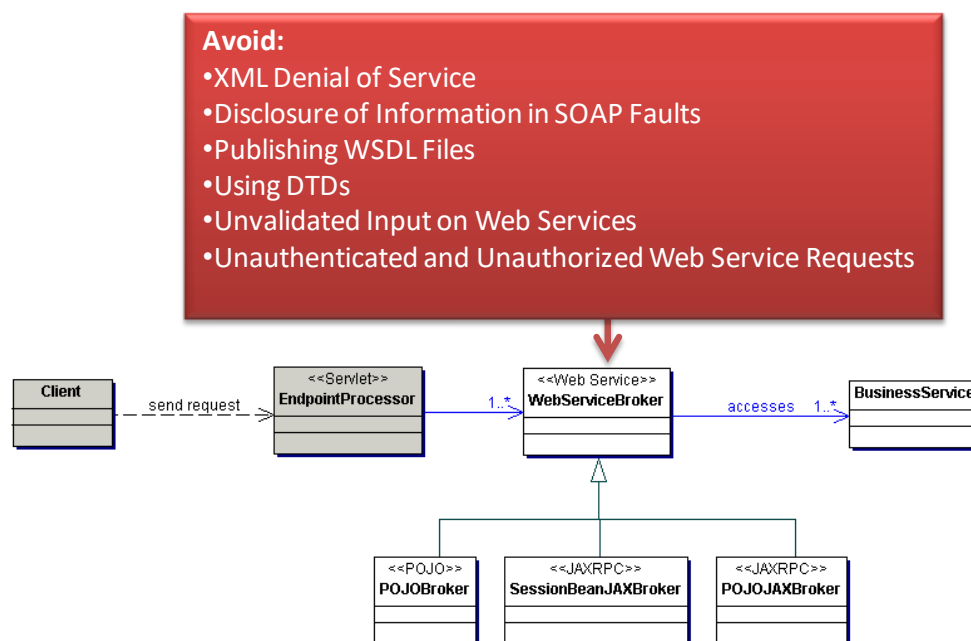
Encrypt database and other server credentials during storage. Unfortunately, any method you use to encrypt the connection string has weaknesses. For instance, WebLogic provides an encrypt utility to encrypt any clear text property within a configuration file^{xx}, but the utility relies on an encryption key stored on the application server. Other methods, such as Jasypt's password based encryption mechanisms require either manual intervention or remote server communication during startup to implement securely^{xxi}. Nevertheless, storing a password plaintext is the least secure option so always use some form of encryption.



WEB SERVICE BROKER

A web service is a popular way of exposing business services to other applications. The integration of different systems, however, can typically involve incompatibilities and complexities. Furthermore, it is desirable to limit the set of services that are exposed by an application as web services. The *Web Service Broker* pattern uses XML and web protocols to selectively expose and broker the services of an application. A `WebServiceBroker` coordinates the interaction among services, collects responses and performs transactions. It is typically exposed using a WSDL. The `EndpointProcessor` class is the entry point into the web service, and processes the client request. The `EndpointProcessor` then invokes the web service through the `WebServiceBroker`, which brokers to one or more services.

DIAGRAM



ANALYSIS

Avoid

XML Denial of Service

Attackers may try Denial of Service (DOS) attacks on XML parsers and validators. Ensure either the web server, application server or an *Intercepting Filter* performs a sanity check on the size of the XML message **prior** to XML parsing or validation to prevent DOS conditions.



Disclosure of Information in SOAP Faults

One of the most common information disclosure vulnerabilities in web services is when error messages disclose full stack trace information and/or other internal details. Stack traces are often embedded in SOAP faults by default. Turn this feature off and return generic error messages to clients.

Publishing WSDL Files

Web Services Description Language (WSDL) files provide details on how to access web services and are very useful to attackers. Many SOAP frameworks publish the WSDL by default (e.g. `http://url/path?WSDL`). Turn this feature off.

Using DTDs

Document Type Definition (DTD) validators may be susceptible to a variety of attacks such as entity reference attacks. If possible, use XML Schema Definition (XSD) validators instead. If a DTD validator is required, ensure that the prologue of the DTD is not supplied by the message sender. Also ensure that external entity references are disabled unless absolutely necessary.

Unvalidated Input on Web Services

Web services often expose critical Enterprise Information Systems (EIS) that are vulnerable to interpreter injection attacks. Protect EIS systems at the web service level with strict input validation on all client-supplied parameters. XML encode untrusted data prior to its inclusion in a web service / XML response.

Unauthenticated and Unauthorized Web Service Requests

Like the other middle and EIS tier components, developers often employ weaker or altogether ignore authentication and authorization controls on web service requests – making web services an ideal target for attackers. Authenticate and authorize every web service request.



REFERENCES

- ⁱ Data Validation, OWASP, http://www.owasp.org/index.php/Data_Validation#Reject_known_bad
- ⁱⁱ Spring Security, <http://static.springframework.org/spring-security/site/>
- ⁱⁱⁱ OWASP ESAPI Project, OWASP, http://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API
- ^{iv} Thread Local object, Java API, <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/ThreadLocal.html>
- ^v Two Security Vulnerabilities in the Spring Framework's MVC, Ryan Berg and Dinis Cruz, http://www.ouncelabs.com/pdf/Ounce_SpringFramework_Vulnerabilities.pdf
- ^{vi} Apache Commons Validator, Apache, <http://commons.apache.org/validator/>
- ^{vii} Security Risks with XSLT, <http://msdn.microsoft.com/en-us/library/ms950792.aspx>
- ^{viii} XPath Injection, Web Application Security Consortium Threat Classification, http://www.webappsec.org/projects/threat/classes/xpath_injection.shtml
- ^{ix} JSTL Technology, Sun Microsystems, <http://java.sun.com/products/jsp/jstl/>
- ^x Java Server Faces Technology, Sun Microsystems, <http://java.sun.com/javaee/javaxserverfaces/>
- ^{xi} Security Risks with XSLT, <http://msdn.microsoft.com/en-us/library/ms950792.aspx>
- ^{xii} XPath Injection, Web Application Security Consortium Threat Classification, http://www.webappsec.org/projects/threat/classes/xpath_injection.shtml
- ^{xiii} WeakReference, Java API, <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/ref/WeakReference.html>
- ^{xiv} Hibernate SQL Injection, OWASP, http://www.owasp.org/index.php/Hibernate-Guidelines#SQL_Injection
- ^{xv} Error Handling, Auditing, and Logging, OWASP, http://www.owasp.org/index.php/Error_Handling%2C_Auditing_and_Logging
- ^{xvi} WS Security Specification, OASIS, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss
- ^{xvii} Internet Message Format, IETF, <http://tools.ietf.org/html/rfc2822#section-3.4.1>
- ^{xviii} Java Secure Coding Standards, CERT, <https://www.securecoding.cert.org/confluence/display/java/FIO34-J.+Ensure+all+resources+are+properly+closed+when+they+are+no+longer+needed>
- ^{xix} Path Traversal Attacks, OWASP, http://www.owasp.org/index.php/Path_Traversal
- ^{xx} Using the WebLogic Server Java Utilities, WebLogic Server Command Reference, http://e-docs.bea.com/wls/docs81/admin_ref/utills17.html
- ^{xxi} Web PBE Configuration, Jasypt, <http://www.jasypt.org/webconfiguration.html>