



Intergral Information Solutions GmbH  
Schickardstr 32 • D-71034 • Böblingen • Germany

**FusionReactor 3**  
**Using FusionReactor Monitoring and Protection**

Doc. Rev. 56, 7 February 2008

---

**Trademarks and Warranties**

FusionReactor, the FusionReactor logotype, and the Integral logotype are trademarks of Intergral Information Solutions GmbH and may not be used without permission. Other trademarks are the property of their respective owners.

The FusionReactor software product, including the FusionReactor JDBC Driver Wrapper is commercial software and may not be redistributed except with the express written agreement of Intergral Information Solutions GmbH. The software may only be used in accordance with the appropriate FusionReactor license agreement.

To the fullest extent applicable by law, Intergral Information Solutions hereby disclaims all warranties, including but not limited to the warranty of merchantability and the warranty of fitness for a particular purpose.

**Feedback**

We welcome feedback on all our products and publications. Please e-mail them to [support@fusion-reactor.com](mailto:support@fusion-reactor.com) and we will address them as quickly as possible.

Published in Germany

Copyright © 2005-2008 Intergral Information Solutions GmbH

All Rights Reserved

---

## Table of Contents

<b>1 Introduction to Protection and Monitoring.....5</b>	
1.1 Scenarios and Scope.....5	
<b>2 Crash Protection Basics.....7</b>	
2.1 Design Goals.....7	
2.1.1 Common J2EE Problems.....7	
2.1.2 Types of FusionReactor Crash Protection.....7	
2.1.3 Survival Strategy.....7	
2.2 Crash Protection Implementation.....8	
2.2.1 Abort Strategies Explained.....8	
Queue.....8	
Abort.....9	
2.2.2 Risks and Restrictions of Request Abort.....9	
JVM Thread Aborts.....9	
Native-Bound Threads.....9	
J2EE Containers and Dead Threads...9	
2.3 The Crash Protection Restrictions Engine 10	
2.3.1 Customer Demographics.....10	
2.3.2 Restrictions Engine Rules.....10	
<b>3 Crash Protection In Action.....11</b>	
3.1 Test Pages.....11	
	3.2 Exercising Timeout Protection.....11
	3.3 Exercising Memory Protection.....12
	3.4 Exercising Request Quantity Protection..13
	3.4.1 Using the 'Queue' Strategy.....14
	3.5 Using the Redirect Method.....14
	3.5.1 Best Practices in Redirection.....15
	Making Marginal Situations Worse....15
	Redirection Isn't Always Possible.....15
	Stuck Threads.....15
	Crash Protection Restrictions.....16
	<b>4 Understanding Crash Protection Restrictions 17</b>
	4.1 Modes.....17
	4.2 Rule Basics.....17
	Request.....18
	Hostname.....18
	Parameters.....18
	Exclude From.....18
	Statistics.....18
	4.3 Examples of Restrictions.....19
	4.3.1 Excluding Batch Jobs.....19
	4.3.2 Including Specific Hosts.....19
	4.3.3 Including a Specific Action Page....20

## Illustrations

Illustration 1: Crash Protection Settings - 8 Seconds.....	11
Illustration 2: Display Message Strategy.....	12
Illustration 3: Request History: Request Timeout.....	12
Illustration 4: Total Memory Available.....	12
Illustration 5: Memory Demand During Test.....	13
Illustration 6: Request History: Request Rejected – Memory Demand Too High.....	13
Illustration 7: Four Requests Total Load.....	13
Illustration 8: Request History: Request Rejected – Load Too High.....	14
Illustration 9: The 'Queue and Notify' Strategy.....	14
Illustration 10: Request History: Request Queued and Expired.....	14

Illustration 11: Redirect Strategy..... 14

# 1 Introduction to Protection and Monitoring

FusionReactor 3 encompasses several technologies designed to ease the burden of J2EE (including ColdFusion) support organizations – two of which are **Enterprise Monitoring** and **Crash Protection**.

**Crash Protection** is, of course, one of the important features of FusionReactor, and has been present since the initial release. FR3 builds on this technology and brings advances in **Restrictions** (which requests will be subject to Crash Protection) and we have also extensively reviewed the mechanisms FusionReactor uses to manage and control requests. Server downtime is often measured in dollars, not minutes, and Crash Protection continues to be a focus of the FusionReactor team.

**Enterprise Monitoring** (available to customers with enterprise licenses) is completely new for FR3, and brings with it exciting new possibilities to visualize the state of a machine cluster, together with behind-the-scenes uptime monitoring, and the ability to send mail when a machine becomes unavailable. Easy-to-read instrumentation including color-coding, real-time graphs and bars, together with advanced grouping, server-offlining and user-definable warning and critical thresholds make our **Enterprise Dashboard** an essential component for hosting providers, or customers with a cluster of J2EE (e.g. ColdFusion) installations.

The Enterprise Dashboard is fully explained in our extensive on-line help, and is the subject of a separate guide, so we won't be covering it here, except where necessary to illustrate how a particular issue is visualized.

## 1.1 Scenarios and Scope

This document will cover the usage of FusionReactor to ensure the availability of your systems. We'll cover Crash Protection and Enterprise Monitoring in some detail. We'll show you how to use FusionReactor to cross-monitor a cluster of machines, such as is typically found within companies with failover solutions, or in hosting providers running large farms of machines.

We'll illustrate in detail, using worked examples, how FusionReactor's sophisticated Crash Protection works to keep your system up and alive, even in the face of an onslaught of requests, bad queries, pages that take too long and memory which is approaching upper margins.

Although every installation of FusionReactor is different – and we're constantly amazed at the diverse and interesting environments FusionReactor is deployed within – we'll also show you how you might tune Crash Protection to optimize your production environments.

Finally, we hope you are happy with FusionReactor. We're convinced it will provide a significant return on your investment, maybe in some cases even paying for itself by preventing one outage. FusionReactor 3 represents significant advances in enterprise monitoring, protection and visualization, but a lot of it would not have been possible without feedback from our customers. We're always grateful for comments on our products; if you have any criticisms or ideas, please send them to:

support@fusion-reactor.com



## 2 Crash Protection Basics

### 2.1 Design Goals

#### 2.1.1 Common J2EE Problems

Crash Protection attempts to keep your server alive by watching for and preventing (or minimizing) the effects of four scenarios, which we identified as being common causes for outages:

- **Requests taking too long** are often an indicator of poor code design. In interactive systems like the web, requests must run in as short a time as possible. It's often preferable to receive an error message rather than continue waiting for a request which may never complete.
- **Too many requests running simultaneously** often leads to resource starvation, and although it's usually possible to tune this using the J2EE engine itself, the system must usually be restarted for it to take effect – and the option is usually well hidden. The majority of J2EE engines – including Macromedia's JRun – control resource access using thread pools, and having too many requests running at once can reduce or exhaust these pools to the state where it's no longer possible to recover the system, leading to a costly restart.
- **Exhausted memory** occurs very frequently in production environments. On shared installations as well as dedicated servers there never seems to be enough memory. Approaching memory margins is a sign that the server is overloaded, or one or more requests are creating a lot of objects and using more than their fair share of storage.

#### 2.1.2 Types of FusionReactor Crash Protection

We designed three crash protection rules to counter these common situations. There's no need to alter any J2EE engine settings to take advantage of these rules, and no container restart is required for them to take effect.

- **Timeout Protection:** Any request which runs longer than a given threshold causes this rule to fire.
- **Request Quantity Protection:** once the number of simultaneous requests reaches this threshold, further incoming requests cause this rule to fire.
- **Memory Protection:** once memory has breached this threshold (specified as a percentage of total), further requests cause this rule to fire.

Using these rules, either individually or in combination, FusionReactor can keep a server alive in a marginal situation for much longer than it would otherwise be available. In the vast majority of cases, Crash Protection rules can keep a server up and responding long enough for the marginal situation to clear, allowing processing to continue normally.

#### 2.1.3 Survival Strategy

Along with each rule, it's possible to specify how FusionReactor will react when that rule fires – the Survival Strategy.

The three available survival strategies are:

- **Abort and notify** – once the rule fires, mail is sent to the administrator, and further incoming requests are aborted either with a fixed message, or a URL redirect to a custom page.

- **Notify** – once the rule fires, mail is sent to the administrator, and the request is allowed to continue normally.
- **Queue and notify** – once the rule fires, mail is sent to the administrator, and the request enters a holding queue until the marginal situation is resolved. The length of the queue – by default 60 seconds – can be configured.

To avoid a flood of email in marginal situations, notification can be turned off completely, or can be set to only send email once in a given period – by default one minute.

All three survival strategies are available for each of the three rules, except **Timeout Protection**, which can't use the **Queue** rule, since the requests it monitors can't be queued once they've started.

## 2.2 Crash Protection Implementation

FusionReactor is implemented as a J2EE Filter and if installed correctly, should be the very first filter in the J2EE chain – this is how the install package installs it, and how our manual installation guide explains it too.

Being the first filter in the chain allows FusionReactor to control exactly what runs within a given J2EE server, and provides the maximum control during marginal situations.

FusionReactor uses its own web server and threading pools to ensure that it does not use J2EE container resources, and therefore also does not rely on an external web server to be available – something that is often one of the first things to become unavailable when resources get tight.

We explicitly designed all aspects of FusionReactor to consume as little time as possible. Turning FusionReactor on should have very little impact on your J2EE application. Part of the product QA cycle involves a multi-day run test under very strenuous conditions to check there are no performance or memory problems.

Activating Crash Protection rules, therefore, can be a routine action; even if you only require a notification and no explicit survival action.

The **Memory Protection** rule is evaluated before the **Request Quantity** rule. If a situation exists in which both rules should logically fire, the **Memory Protection** rule (often the more dangerous of marginal situations) is given preference to deal with the situation.

### 2.2.1 Abort Strategies Explained

Apart from the **Notify** action, which sends a detailed email during Crash Protection interventions, there are two other major strategies for dealing with requests: **Abort** and **Queue**.

#### Queue

This strategy attempts to alleviate the marginal situation by temporarily pausing incoming requests until the condition which caused the rule to fire no longer exists.

The length of the queue can be specified, after which requests will be terminated regardless of the survival strategy.

There is no limit on the size of the queue, so if a large quantity of requests are present they will consume tracking resources inside FusionReactor (albeit temporarily). This strategy is therefore best used on systems with sufficient memory, or on systems where the volume of requests is known and not expected to become prohibitive.

### Abort

This strategy aborts requests. When used with the **Memory** or **Request Quantity** rules, it is best imagined as a 'reject', rather than an abort. The request is summarily rejected and not allowed to proceed inside the J2EE engine. The abort strategy – redirect to URL or display of fixed message – is applied prior to the abort action.

When used with the **Timeout Protection**, FusionReactor uses strong thread manipulation techniques to make sure requests are stopped.

In all abort actions, FusionReactor will process the terminating requests for statistical and tracking purposes.

## 2.2.2 Risks and Restrictions of Request Abort

### JVM Thread Aborts

Because a request abort is a last-ditch effort to prevent a marginal situation become critical, FusionReactor attempts to use a rather strong method to halt processing.

In the vast majority of these cases, the thread stops immediately and FusionReactor is able to recover the system.

However, in a very small set of theoretical cases – and we must emphasize we've never seen this happen during innumerable test runs – the JVM could be restarted. Most J2EE service wrappers will automatically restart it in this case, causing a minor outage – which would in any case be less seriously than a hung system which would otherwise occur.

In the light of our test results, and the use of request abort in our own production environments, we are still utterly convinced that the benefits of this strategy far outweigh the infinitesimally small risk.

### Native-Bound Threads

Due to restrictions in the JVM, it is not always possible to immediately stop a thread. If a thread is currently engaged in a blocking native operation, i.e. performing I/O in a JNI method (sockets are a good example of this) then the JVM will not be able to abort the thread until it completes.

For this reason, you may not see requests disappear immediately when aborted with Crash Protection or killed manually from the FusionReactor Administrator.

We're continuing to investigate this problem – which is a restriction of the Java virtual machine – and will release an updated version of FusionReactor when we have a solution.

### J2EE Containers and Dead Threads

After aborting a thread, FusionReactor processes the requests for statistical and display purposes inside the FusionReactor Administrator, then disposes of the original thread object by returning it to the J2EE engine.

Although the object no longer represents a valid thread, some engines do not check this, assuming the thread to still be runnable, and return the object back to their internal thread pool.

In some cases, this object can then be picked by the pool to run a new incoming request. This will then fail immediately, at which point the engine will remove the thread from the pool. The client then sees an error message, usually accompanied by a HTTP status 500 – Internal Server Error.

Again, this happens infrequently and is vastly preferable to a total outage.

## 2.3 The Crash Protection Restrictions Engine

This section illustrates a brand new feature to FR3 – **Crash Protection Restrictions**.

This powerful filter is used to configure exactly which requests will benefit from Crash Protection. There are certain classes of request – batch jobs, database update, data warehousing – which are known to consume time and resources and which could be safely ignored by Crash Protection, so-called 'false positives'.

### 2.3.1 Customer Demographics

During the design of this feature, we identified two key demographics for whom this would be an important technology:

- The small homepage or small-company site which is agile and which will protect all requests **except** its long-running updates.
- The large enterprise site, possibly with many web services and non-interactive services, which will be default **not** protect requests, preferring to define small groups of interactive pages which will be protected.

Instead of attempting to aggregate these groups into those who would and those who wouldn't use the feature, we made the Restrictions Engine reversible. The engine – which is totally optional – can be run in two distinct modes:

- All requests are protected **except** those matched by a filter rule (*exclude mode*).
- All requests are ignored **except** those matched by a filter rule (*protect mode*).

We feel these two modes accurately reflect the needs of the two demographics above.

### 2.3.2 Restrictions Engine Rules

In order to decide whether a given request will (or will not) be protected, the Restrictions Engine evaluates a number of user-defined rules.

Each rule allows fine-grained specification of exactly which requests it matches.

A rule is specified as an exact match or a Regular Expression as defined by the relevant Java JRE specification<sup>1</sup>. As well as the URL path, the following optional components can be used to define the rule:

- The **requested hostname** – useful to differentiate requests on multi-homed systems
- The **URL parameters** – useful for requests whose behavior changes according to their parameters

Additionally, each rule can specify whether pages matched by that rule will be protected by (or excluded from) all protections or just Timeout protection. If the list is running in exclude mode, in which all requests are protected by default, it's also possible to specify whether any excluded requests will still be tracked for statistical purposes (e.g. runtime tracking etc.).

---

<sup>1</sup> For JRE 1.4, this is defined [here](#).

## 3 Crash Protection In Action

This section gives practical examples of how Crash Protection can be used. It will illustrate how FusionReactor tracks Crash Protection actions and how you can exercise the engine to try out your rules.

### 3.1 Test Pages

In order to exercise the three types of Crash Protection, we have written four pages (source code is presented in the appendix):

- **pause.cfm** – takes a 'timeout' parameter (in seconds). This page simply sleeps for the given number of seconds before completing. It's used to exercise the **Timeout Protection**.
- **load.cfm** – takes a 'load' and 'timeout' parameter. This page loads the system by spawning itself to produce the required number of requests. Each page then sleeps for the given number of seconds before completing. It's used to exercise the **Request Quantity Protection**.
  - E.g. Calling `load.cfm?load=5&timeout=10` would cause 4 requests to be spawned (giving a total load of 5), and each one would sleep for 10 seconds.
- **grow.cfm** – takes a 'freemem' parameter (MB) and allocates memory continually until that threshold is reached, then sleep for 60 seconds.
- E.g. Calling `grow.cfm?freemem=10` would instruct the page to allocate memory until there was only 10 MB remaining, then sleep for 60 seconds.
- **system.cfm** – simply outputs the version of Java in use. This page is used as a 'hello world'-type request to see if the system will accept requests.

These pages are placed in a temporary folder in our test machine. We use the following URL to access them:

```
http://int0234/tmp/ts1/pause.cfm?timeout=10
```

### 3.2 Exercising Timeout Protection

In the FusionReactor Administrator, access the CP Settings area by selecting it in the contents area. You may need to open the 'Crash Protection' section to see this.

The screenshot shows the 'Crash Protection' configuration interface. Under the 'Timeout Protection' section, the 'Timeout Protection (seconds)' field is set to 8, and the strategy is 'Abort and Notify'. Below this, the 'Protection Restrictions' section shows 'Restrictions' set to 'Disabled' and 'Behavior' set to 'Exclude requests that match the rules'.

Illustration 1: Crash Protection Settings - 8 Seconds

Once the crash protection settings page has loaded, ensure the Restriction engine is disabled, and timeout protection is set to 8 seconds, with the **abort and notify** strategy. Illustration 1 shows this setting.

Once this has been committed with the **Save Crash Protection Settings** button, the Timeout Protection becomes active. We can then run the pause script to show the protection in action.

In our browser, we use the url `pause.cfm?timeout=10` to ensure that the script is still running at the timeout protection limit of 8 seconds. In fact, since this is the first time the page has been run, ColdFusion will require a short period to compile it first. This additional time is also counted and tracked as part

of the 8-second protection limit. The page would therefore run for longer than the 10-second pause time, giving the Timeout Protection ample opportunity to activate.

When the protection activates, the **Display Message** abort strategy is used, and the user sees an appropriate message. The abort strategy and the display message are both configurable using the Crash Protection Settings screen, which we used earlier to set up the protection. Our **pause.cfm** script simply uses a `Java Thread.sleep()` call in order to go to sleep, so it is easy for FusionReactor to wake up this thread prior to killing it. As we noted earlier though, if the request was performing socket I/O, FusionReactor would be forced to wait until the script returned from the socket operation before it could terminate it. Most operations can be terminated immediately though.

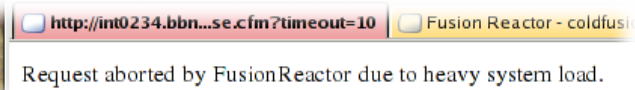


Illustration 2: Display Message Strategy

The **Request History** page shows the abort by annotating it with a 'timeout' flag, as illustrated X.

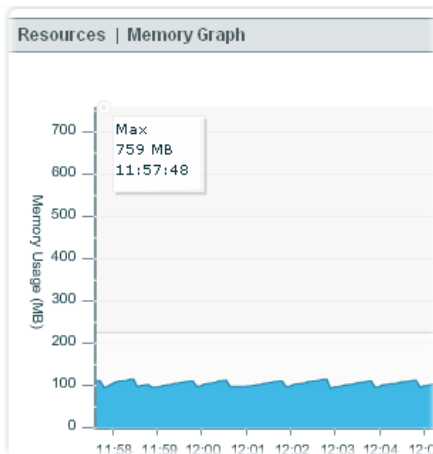


Illustration 3: Request History: Request Timeout

### 3.3 Exercising Memory Protection

Exercising memory protection is a little more difficult, due to the disparate memory configurations in use, but here's how we use FusionReactor to show it in action in our test environment.

Illustration 4: Total Memory Available



The first step is to work out how much memory is available to us. Instead of hunting through J2EE configurations, we can use FusionReactor to tell us. Access the **Memory Graph** (accessible from the table of contents under the 'Resources' section) and have a look at the figure at the top left of the Y axis (see Illustration 4 for our value – about 203 MB).

Memory Protection uses a percentage value as the upper limit, to ensure a marginal situation does not become critical. If we define a Memory Protection of 10%, this translates to about 20 MB. The actual effect of this protection will be to reject new requests when the quantity of free memory drops below 20 MB. During this test, we'll be using the **Memory Graph** to observe the situation.

In the Crash Protection Settings page, we set up the **Memory Protection** to activate at 10%, and make sure any value in the **Timeout Protection** box is removed. We commit the change using the **Save Crash Protection Settings** button, as before.

The system is now primed for the test. We use the **grow.cfm?freemem=8** script to run the system to 195 MB total memory demand (about 8 MB free memory), a value which is well within the 20 MB margin (which occurs at 180 MB).

Any requests which occur while the system is in this marginal situation will be rejected by the **Memory Protection**. We can watch the **Memory Graph** to observe the memory load during the test. When the **grow.cfm** page reports it has exhausted memory to the required limit, we run the **system.cfm** script, which is immediately rejected. Illustration 11, below, shows the memory graph during the test. Shows the memory load during the test, including the calm period before the test, and the demand peak caused by **grow.cfm**.

The sawtooth pattern before and after the test is a classic example of Java's periodic garbage collection in action. The memory demand caused by **grow.cfm** isn't actually released back to the system until a few seconds after the request completes – again, this is normal since Java only reclaims this memory when it's urgently required, or in this case, when system activity declines to a calm level.

By observing the **Request History** page (Illustration 6, below), we can see that FusionReactor correctly rejected the **system.cfm** request during the marginal situation, and that it's been tagged with a 'rejected' flag.

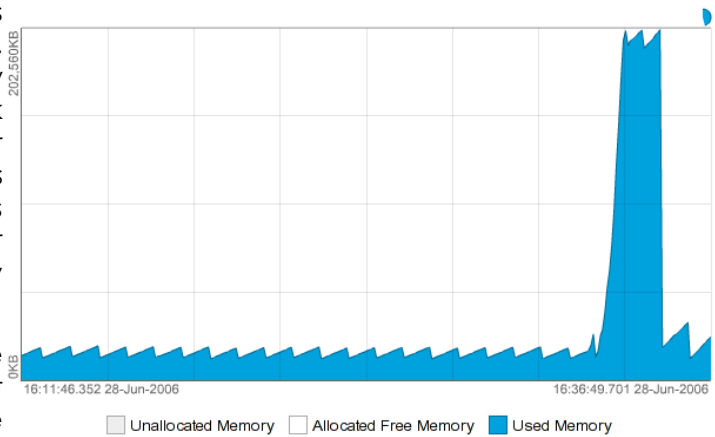


Illustration 5: Memory Demand During Test

16:34:58.161	200 Rejected	9	http://in0234.bbn.integral.com/tmp/ts1/system.cfm	10	Cur:(99%)201,425
28-Jun-2006	192.168.0.74	jrpp-4			Free:1,134

Illustration 6: Request History: Request Rejected – Memory Demand Too High

### 3.4 Exercising Request Quantity Protection

The Request Quantity Protection rejects incoming requests when the total request load on the system exceeds a certain value. This protection is completely independent of any values defined by your web server or J2EE container. Like other FusionReactor protections, this value can be changed without causing a restart of any other software components, minimizing outage time.

To exercise this protection, access the Crash Protection Settings page using the table of contents, and remove any other protections which may be left over from previous tests.

Set a **Request Protection** value of **4** and commit the changes using the **Save Crash Protection Settings** button. When the number of running requests rises to 4, FusionReactor will control the load by rejecting any incoming requests.

The screenshot shows the 'Running Requests (4)' page in FusionReactor. It includes a table with columns for Started, IP, ID, URL/Parameters, Time(ms), and Memory(KB). There are also buttons for 'Stack Trace ALL' and 'Kill ALL', and memory usage statistics.

Started	IP	ID	URL/Parameters	Time(ms)	Memory(KB)
17:08:00.545 28-Jun-2006	192.168.2.52	13 jrpp-9	http://in0234.bbn.intergral.com/imp/1/load.cfm? timeout=120	2,684	Cur:(14%)28,548 Free:174,011
17:07:59.534 28-Jun-2006	192.168.2.52	12 jrpp-8	http://in0234.bbn.intergral.com/imp/1/load.cfm? timeout=120	3,695	Cur:(14%)28,469 Free:174,090
17:07:58.482 28-Jun-2006	192.168.2.52	11 jrpp-7	http://in0234.bbn.intergral.com/imp/1/load.cfm? timeout=120	4,747	Cur:(13%)27,828 Free:174,731
17:07:57.721 28-Jun-2006	192.168.0.74	10 jrpp-6	http://in0234.bbn.intergral.com/imp/1/load.cfm? load=4&timeout=120	5,508	Cur:(11%)23,409 Free:179,150

Illustration 7: Four Requests Total Load

Using the **load.cfm?load=4&timeout=120** script, we load the system with 4 concurrent requests for 120 seconds, visible in the **Running Requests** page, as shown in Illustration 7, above.

Running the **system.cfm** page while this load exists causes the page to be rejected, as shown in Illustration 8.

The screenshot shows a single entry in the 'Request History' table. The status is 'Rejected' in red text. The URL is 'http://in0234.bbn.intergral.com/imp/1/system.cfm'.

17:08:13.414 28-Jun-2006	200 Rejected 192.168.0.74	14 jrpp-10	http://in0234.bbn.intergral.com/imp/1/system.cfm	301	Cur:(8%)14,036 Free:188,523
-----------------------------	------------------------------	---------------	--	-----	--------------------------------

Illustration 8: Request History: Request Rejected – Load Too High

### 3.4.1 Using the 'Queue' Strategy

We will also use this scenario to demonstrate the 'queue' survival strategy. Any requests which arrive are queued by FusionReactor to give the system time to recover. The default queue period is 60 seconds, which is usually ample time.

Use the **Crash Protection Settings** page to change the survival strategy for **Request Protection** to **Queue and Notify**, as shown in Illustration 9. We'll leave the queue timeout at 60.

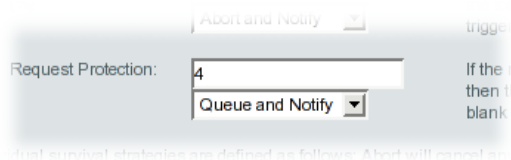


Illustration 9: The 'Queue and Notify' Strategy

Firstly, we'll show requests being queued and then completing. Running the **load.cfm?load=4&timeout=60** script, we increase the system request load to the Crash Protection limit of 4. Running the **system.cfm** we see the page pause until the **load** pages complete and the total load reduces below the margin.

If a page remains in the queue longer than the **Queue Timeout** period, it will be rejected by the protection. We can illustrate this by running the **load.cfm?load=4&timeout=120**, and then running the **system.cfm** script. The load will be applied for 120 seconds, which is longer than the queue period. The **system.cfm** script will then be **expired** from the queue. Illustration 10

shows the **Request History** detail for an expired page. The page expired from the queue after 60,797ms.



	17:26:40.105	200 Expired	29	http://int0234.bbn.intergral.com/tmp/ts1/system.cfm	60,797	Cur: (8%) 17,317
	28-Jun-2006	192.168.0.74	jrpp-16			Free: 185,242

Illustration 10: Request History: Request Queued and Expired

### 3.5 Using the Redirect Method

Up to this point, we have been using the abort strategy to return a short message to the client whenever a marginal situation arises. While this is usually sufficient to warn a client that the system is currently recovering, there are cases where customers may want to return their own branded pages to the user.

The Crash Protection system therefore also allows an HTTP **Temporary Redirect** message to be sent to the client. This is configured on the **Crash Protection Settings** page in the **Survival Strategy** section, as shown in Illustration 11. When a marginal situation arises which would otherwise cause a page to be aborted or rejected, FusionReactor will now attempt to send an HTTP redirect to the client. If the 'Add Parameters' setting is enabled, FusionReactor will also supply some information as to the cause of the redirect. This information can be used in customers' own scripts to provide custom branded error messages.



Illustration 11: Redirect Strategy

If enabled, parameters are passed as URL (GET) variables and are named as follows:

- **DETECTION\_METHOD**  
One of **timeout**, **freemem** or **requests**, specifying the protection which fired.
- **THRESHOLD\_VALUE**  
The threshold value associated with this protection.
- **ACTUAL\_VALUE**  
The actual value which caused this rule to fire.

For the **timeout** method, the threshold and actual values are specified in milliseconds. For the **freemem** method, the values are in megabytes.

#### 3.5.1 Best Practices in Redirection

We believe redirection is a great way to provide branded error output to clients, but there are a couple of caveats to be aware of.

##### Making Marginal Situations Worse

If you target a redirection to a page which resides within the J2EE container that FusionReactor is currently monitoring, you may make a marginal situation worse by providing additional load.

Using a redirection target in the same container is possible - indeed this may be your only option - so try to design target pages which place low demands on the server:

- Few images
- Low memory demand from J2EE scripting languages (JSP, CF etc.)

- Low processor demand – no database work, no CPU-intensive computation

Design of redirection targets should therefore be a compromise between corporate branding guidelines and server load.

### **Redirection Isn't Always Possible**

FusionReactor uses an HTTP Temporary Redirect to instruct the client to display the redirection target page. This is the safest, fastest way of sending a redirect. It guarantees the returned page is valid, and produces low server overhead in memory and CPU cycles.

However, this redirection method can *only* be used if the HTTP response headers have not yet been committed. If a page has already begun streaming body data, the thread will be aborted as normal, but the redirection will not occur.

If page design takes this into account, by performing business logic before display logic, this is not likely to become a problem.

### **Stuck Threads**

If your redirection target lies within the same J2EE container which caused the redirection, the problem outlined in J2EE Containers and Dead Threads (see page 9) can occur. If the J2EE container picks the newly-stopped thread to serve your redirection target without first checking that the thread is runnable, a 500 Server Error can occur.

### **Crash Protection Restrictions**

Finally, you may like to add your redirection target as a rule in the **Crash Protection Restrictions** engine, in order that it will be ignored by Crash Protection if it occurs.

## 4 Understanding Crash Protection Restrictions

This section explains the **Crash Protection Restrictions Engine**, one of the most powerful features in FusionReactor.

### 4.1 Modes

#### Crash Protection Restrictions

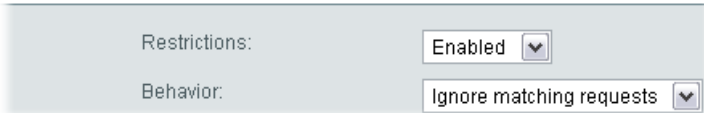


Illustration 12: Enabling Crash Protection Restrictions

The Restrictions Engine decides which requests will be monitored by Crash Protection, and which requests will be allowed to proceed unobserved. The engine is configured in FusionReactor Administrator in the **Crash**

**Protection Restrictions** section of the **CP Settings** page. Any changes you make to these settings, as well as any changes you make to individual rules, become active immediately without necessitating any software restarts. This allows you to test and tune rules 'on the fly'.

The engine operates in one of three modes:

- **Disabled** – in which the engine is completely bypassed. In this mode, all requests are monitored by Crash Protection if any protection is currently active.
- **Ignore matching requests** – in which the engine will, by default, protect all requests. Any requests which match a rule are not monitored.
- **Protect matching requests** – in which the engine will, by default, ignore all requests. Any requests which match a rule **will** be monitored.

### Engine Overhead

One of the chief design goals of the Restrictions Engine is that it demand very low overhead during the course of a request. The engine is optimized for very low CPU and memory demand and is almost undetectable when active. The Restrictions Engine can therefore be used even on very busy systems.

### 4.2 Rule Basics

Rules for the Restrictions Engine are manipulated using the **CP Restrictions** item in the table of contents. The settings page allows you to then add, remove and change any rules currently entered into the engine.

Rules are evaluated from top to bottom, and the first one which fires causes the engine to stop evaluating any further rules.

The Restrictions Engine matches each rule against components of the request URI. The exact components used during the match are selectable using the fields on the rules form (see Illustration 13).

The exact options available depend on the current mode of the engine: in **Protect** mode, the **Statistics** option is not available – all requests which match a

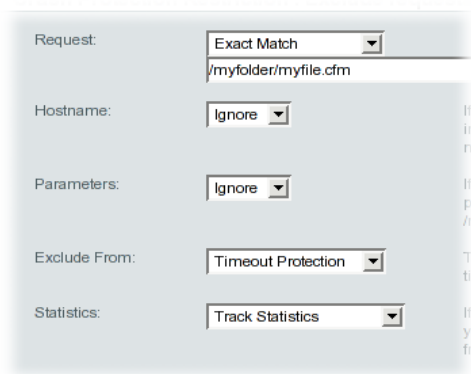


Illustration 13: Configuring Restrictions

rule are tracked for timing statistics. The labels used also change to reflect the mode of the system, and the make the meaning of the rule easier to understand.

### Request

These two fields define the match mode of the rule, and the actual match string. The drop-down box specifies whether the text field is an exact string match, or whether the field contains a Java Regular Expression.<sup>2</sup>

The remaining options tell the engine exactly which request fields to match against the entered string.

### Hostname

If enabled, this field specifies that the string begins with a **hostname**. This allows rules to target specific pages when addressed by multiple sites. The value of this field should match the HTTP 1.1 'Host' header. No scheme (`http://` or `https://`) should be applied.

This header is used by web servers, browsers and J2EE containers to differentiate requests for multiple websites which may reside on the same physical system.

A later section will illustrate how to use this feature to protect only certain page requests.

### Parameters

This setting specifies whether FusionReactor will differentiate pages based on their URL (GET) parameters. A common design pattern is to change the behavior of a request based on the information provided as URL variables.

An example of this might be a **doAction.jsp** page, whose action is specified as a parameter. Some behaviors of this page should be exempt from protection (**doAction.jsp?action=PopulateDataWarehouse**, for example), while some should be observed and tracked (**doAction.jsp?action=ServeFile**).

This setting allows FusionReactor to treat the two requests separately, and decide whether to protect them based on the URL parameters.

### Exclude From

(in **Protect** mode, this field is called **Protection Type**).

Specifies which specific protection is affected by this rule.

- **If the engine is in Exclude mode**, this field specifies which protections any matching requests will be exempt from – either **Timeout Protection** or **All Crash Protection**. If exempt from only Timeout Protection, a matching requests will still be protected by Request Quantity Protection (i.e. will be rejected if the request load is too high) and Memory Protection (i.e. will be rejected if the memory demand is too high). If exempt from **Timeout Protection**, requests will be allowed to run to completion, provided neither the Memory nor Request Quantity protections are activate.
- **If the engine is in Protect mode**, this field specifies which protections will apply to any matching requests – either **Timeout Protection** or **All Crash Protection**. Any requests which don't match will proceed into the J2EE container unprotected.

### Statistics

(only available in **Exclude** mode)

---

<sup>2</sup> JRE 1.4 Regular Expressions are defined [here](#).

If a request matches a rule and is therefore excluded from protection, this setting specifies whether its timing values will still contribute to FusionReactor metrics.

## 4.3 Examples of Restrictions

The restrictions engine, though powerful, can be overwhelming at first. This section provides some worked examples to illustrate the different modes and protections.

### 4.3.1 Excluding Batch Jobs

This example excludes our named batch jobs from timeout protection.

<b>Active Protections</b>	Timeout Protection @ 8 seconds	
<b>Engine Mode</b>	Exclude	
<b>Rule</b>	Exact match	/scripts/CleanUpDatabase.jsp
	Exclude from	Timeout Protection
<b>Page Decisions</b>	/scripts/CleanUpDatabase.jsp? db=MyDatabase	<b>Ignored</b>

If we want to exclude this page anywhere it occurs, we can use a regular expression.

<b>Active Protections</b>	Timeout Protection @ 8 seconds	
<b>Engine Mode</b>	Exclude	
<b>Rule</b>	Regular Expression	(.*)CleanUpDatabase\.jsp
	Exclude from	Timeout Protection
<b>Page Decisions</b>	/scripts/CleanUpDatabase.jsp? db=MyDatabase	<b>Ignored</b>
	/bigsite/jspscripts/CleanUpDatabase.jsp	<b>Ignored</b>

Similarly, if all our batch scripts were named **batch\*.jsp**, we could also ignore them with an appropriate regular expression.

<b>Active Protections</b>	Timeout Protection @ 8 seconds	
<b>Engine Mode</b>	Exclude	
<b>Rule</b>	Regular Expression	(.*)batch(.*)\.jsp
	Exclude from	Timeout Protection
<b>Page Decisions</b>	/scripts/CleanUpDatabase.jsp? db=MyDatabase	<b>Ignored</b>

### 4.3.2 Including Specific Hosts

If we wanted to include a specific host in Crash Protection only, the following rule might suffice (NB the alias "testvm234" points to the same machine as the first URL).

<b>Active Protections</b>	Timeout Protection @ 8 seconds	
<b>Engine Mode</b>	Protect	
<b>Rule</b>	Regular Expression	Int0234.bbn.intergral.com/(.*)
	Protection Type	All Crash Protection
	Hostname	Check
<b>Page Decisions</b>	http://int0234.bbn.intergral.com/testPage.jsp	<b>Protected</b>
	http://testvm234.bbn.intergral.com/testPage.jsp	<b>Ignored</b>

### 4.3.3 Including a Specific Action Page

The following rule specifies an action page with many parameters which normally takes a few minutes to complete when run in a certain mode. We exclude it from Timeout Protection.

<b>Active Protections</b>	Timeout Protection @ 8 seconds	
<b>Engine Mode</b>	Exclude	
<b>Rule</b>	Regular Expression	(.*)mightyActionPage\.jsp(.*)action=backupDb(.*)
	Protection Type	Timeout Protection
	Parameters	Check
<b>Page Decisions</b>	http://int0234.bbn.intergral.com/testPage.jsp	<b>Protected</b>
	http://testvm234.bbn.intergral.com/testPage.jsp	<b>Ignored</b>