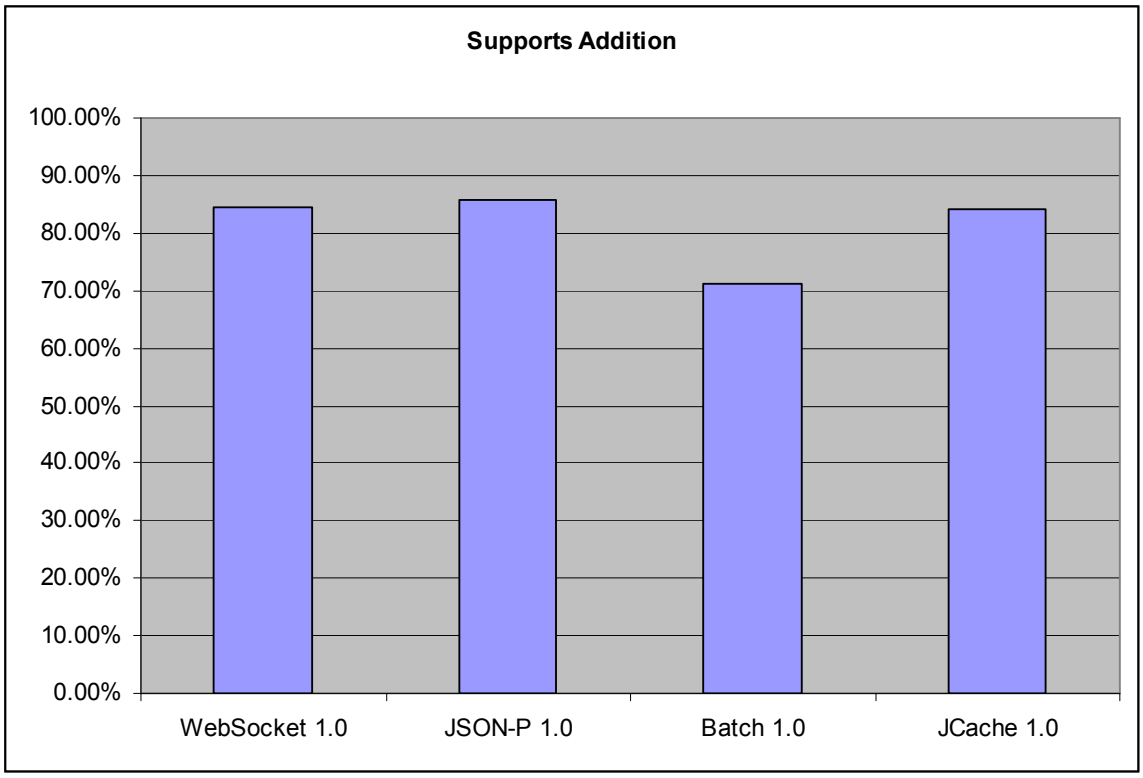
Java EE 7 Survey Summary

There were a large number of developers that participated in the survey -- over 1,100. The results were probably not all that surprising but does offer some good insights, especially from the large number of thoughtful comments to almost every question asked.

APIs to Add to Java EE 7 Full/Web Profile

There was significant support for adding all the new APIs to the full profile. The following graph shows the percentage of people that wanted to add each new API to the platform:



As the graph depicts, support is relatively the weakest for Batch 1.0, but still good. A lot of folks saw WebSocket 1.0 as a critical technology with comments like:

- "WebSocket must be a priority"
- "A modern web application needs Web Sockets as first class citizens"
- "WebSocket is very important"
- "At least WebSocket should be included in to Web Profile"

While it is clearly seen as being important, a number of commenters expressed dissatisfaction with the lack of a higher-level JSON data binding API:

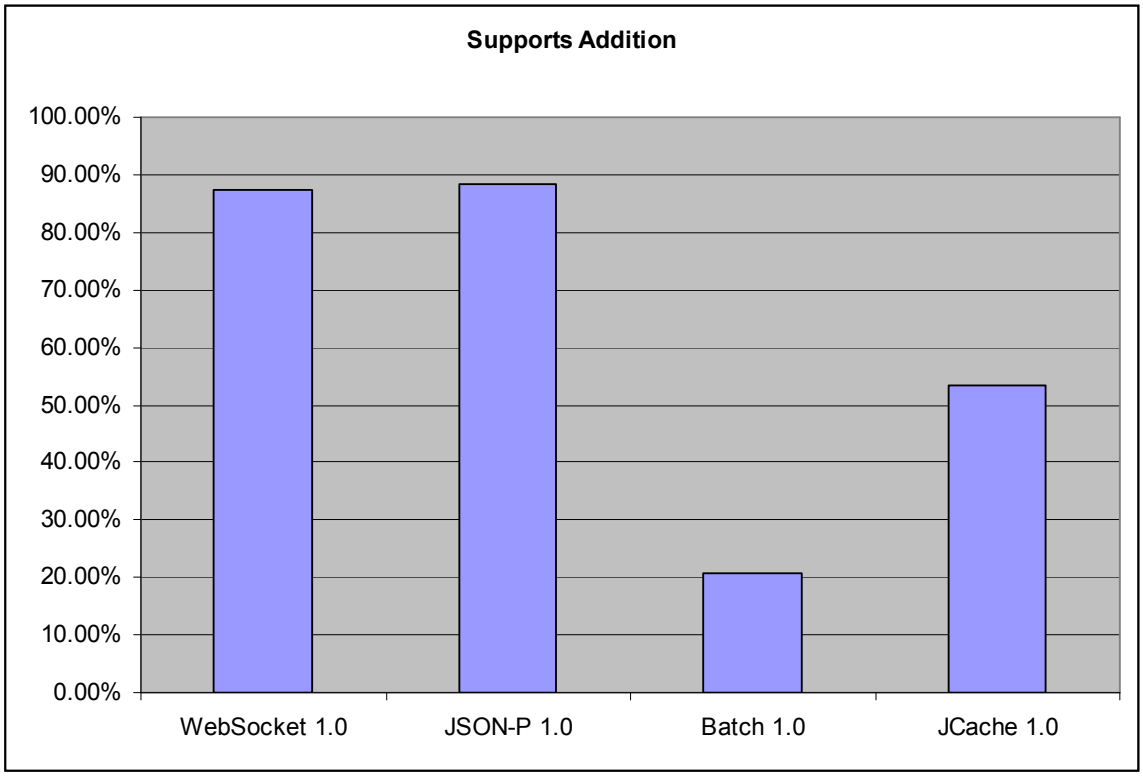
- "How come we don't have a Data Binding API for JSON"

- "Without Data Binding for the JSON API I think the API will not be useful for many developers"
- "For JSON-P I said 'not sure' because the status of the API is definitely #@!%"

JCache was also seen as being very important as expressed with comments like:

- "A standard in caching...will go a long way towards enriching the language"
- "JCache is very important"
- "JCache and WebSocket feels incredibly important for both"
- "JCache should really be that foundational technology on which other specs have no fear to depend on"

The results for the Web Profile is also perhaps somewhat predictable. While there is strong support for adding WebSocket 1.0 and JSON-P 1.0 to the Web Profile, support for adding JCache 1.0 and Batch 1.0 is relatively weak. There was actually significant opposition to adding Batch 1. 0 (with 51.8% casting a 'No' vote).



Enabling CDI by Default

A significant majority of 73.3% developers supported enabling CDI by default. Only 13.8% opposed enabling. The comments reflect a strong general support for CDI as well as a desire for better Java EE alignment with CDI:

- "Why require an empty beans.xml if you normally always want to use CDI?"
- "Yes, every archive found should be eligible to be a bean archive"
- "Yes, Yes, Yes!"
- "All EJB and JSF managed beans functionalities should be replace by CDI"
- "CDI makes EE quite valuable"
- "It would be better if CDI is there by default since an empty beans.xml doesn't make sense at all"
- "CDI should become the cornerstone of Java EE!"
- "I think so because dependency injection is an absolute necessity in modern day app development"
- "Would prefer to unify EJB, CDI and JSF lifecycles"
- "Absolutely yes, please!"
- "I really think so because most of the applications use beans.xml only to enable CDI"
- "CDI is incredible technology"
- "It is about time that everyone got the benefits of dependency injection"
- "In fact CDI should replace EJB in the long run"
- "DI is a must in modern Java applications, having to turn it on explicitly is a huge drawback"

There is, however, a palpable concern around the performance impact of enabling CDI by default:

- "Java EE projects in most cases use CDI, hence it is sensible to enable CDI by default when creating a Java EE application. However, there are several issues if CDI is enabled by default: scanning can be slow - not all libs use CDI (hence, scanning is not needed)"
- "CDI scanning adds to the initialization process"
- "This can turn class scanning slower and I don't think it is error prone"
- "If it can be done without affecting deployment time, then yes"
- "Yes, but if there is no performance problem"

Another significant concern appears to be around backwards compatibility and conflict with other JSR 330 implementations like Spring:

- "I am leaning towards yes, however can easily imagine situations where errors would be caused by automatically activating CDI, especially in cases of backward compatibility where another DI engine (such as Spring and the like) happens to use the same mechanics to inject dependencies and in that case there would be an overlap in injections and probably an uncertain outcome"
- "Since there are other solutions not sure if this won't create issues for developers that plan to use DI solutions other than CDI"
- "This would break backwards compatibility"

Some commenters attempt to suggest solutions to these potential issues:

- "If you have Spring in use and use javax.inject.Inject then you might get some unexpected behavior that could be equally confusing. I guess there will be a way to switch CDI off. I'm tempted to say yes but am cautious for this reason"
- “It should be manually deactivated if not needed”
- “but we should have an option to disable it”
- “Provided we have a way of disabling CDI for a jar”
- “I don't like the way that we enabled CDI now by adding a beans.xml file to our application. What is the impact to enabling it by default? Performance? If there's no significant impact, why not by default enable CDI, but allow people to disable it using the beans.xml file?”

Consistent Usage of @Inject

A slight majority of 53.3% developers supported using @Inject consistently across JSRs. 28.8% said using custom injection annotations is OK, while 18.0% were not sure. The vast majority of commenters were strongly supportive of CDI and general Java EE alignment with CDI:

- "Use only @Inject without API specific injection annotations"
- "CDI is made for this job"
- "No, except in cases where they really need something significantly different or specific, or the resulting standardized syntax is just terrible"
- "Standards are good because they keep the learning curve down"
- "One annotation to define them all!"

- "@Inject should be enough. Different annotations make using JSRs more difficult and gives a feeling of 'yet another mechanism'"
- "I think to be consistent, JSR-330 should be followed throughout"
- "If we make DI central we can solve related bugs at one place and not spread issues through JSRs"
- "Stick to @Inject!"
- "It's already bad enough that the same scopes are defined by multiple specs (e.g. JSF 2.0 and CDI 1.0). Simplify the programming experience by using one common approach, preferably CDI"
- "Dependency Injection should be standard from now on in EE. It should use CDI as that is the DI mechanism in EE and is quite powerful. Having a new JSR specific DI mechanism to deal with just means more reflection, more proxies. JSRs should also be constructed to allow some of their objects Injectable. @Inject @TransactionalCache or @Inject @JMXBean etc...they should define the annotations and stereotypes to make their code less procedural. Dog food it. If there is a shortcoming in CDI for a JSR fix it and we will all be grateful"
- "Keep to standards specified, we do not want any Einsteins doing things they should not"
- "I'd rather vote no, but what about those specs that already had invented their own annotations? Should they be migrated to @Inject, or better yet support both?"
- "Please build on CDI!"
- "There are already too many annotations all over the place. Can you imagine having many variations of the new operator coming from different JSRs/libraries! One annotation to inject managed object is the goal to achieve"
- "More closely following CDI @Inject would be preferred to encourage consistency across technologies"
- "Verbosity is acceptable as compared to variety of standards. We need uniformity over variety as it makes the API maintainable and easier to learn"
- "Absolutely not!"
- "There is no good reason for multiple ways to do something so conceptually simple"
- "I think that non-CDI injection mechanisms should be avoided and possibly deprecated, when they already exists. However, one reason to implement your own CDI could be

compatibility with SE or Servlet containers, so it may be worthwhile sometimes to fallback to non-CDI injection, when CDI is not available"

- "IMO having multiple injection mechanisms is currently quite confusing... should I stick to @EJB, or use @Inject? What are the differences? How should a newbie find it out?"
- "Even with a little extra verbosity it is better to use CDI syntax. Using your own injection annotations per technology easily leads us to different syntax per technology"
- "A uniform 'best way' introduces clarity"
- "We do not want to overrun developers with too many annotations to remember. @Inject should be the standard annotation for performing injection"
- "We're trying to make this a comprehensive platform, right? Injection should be a fundamental part of the platform; everything else should build on the same common infrastructure. Each-having-their-own is just a recipe for chaos and having to learn the same thing 10 different ways"
- "If CDI cannot be used or evolved to cover all the cases that other APIs require in a flexible fashion, it shouldn't be enforced"
- "I think that new Java EE technologies should follow CDI closely"
- "It's extremely hard to understand that different injection annotations cause similar behavior for Java EE beginners"
- "I would even recommend to replace existing EE annotations by CDI"
- "But CDI @Inject has to be robust enough not to hold back innovation/updates to the new EE technologies"
- "One standard!"
- "Dependency injection is tough enough to explain to newbies without obfuscating it with half a dozen inconsistent syntaxes"
- "Multiple different annotations for injection causes lots of confusion (as can be seen at StackOverflow.com and other forums). Having one way of doing it, though more verbose, would simplify matters. I dare say it should be taken a bit further and the question and answer be applied more broadly. Anytime there is overlapping functionality between JSRs, it should be branched out and standardized if it's something as central or integral such as injection"
- "Use @Inject everywhere!"

- "One thing to rule them all: CDI @Inject ;-)"
- "Pick a standard, stick to it"
- "Having uniformity will lead to an easier learning path and thereby keeps the platform developer friendly"
- "The Java EE platform definitely needs more alignment. I understand that everyone also wants their spec to function standalone, but fragmentation in how to do things is a very frequent complaint against Java EE (e.g. like CDI beans vs EJB beans vs JSF Managed Beans, etc). CDI should make it as easy as possible to be added to Tomcat, but at least for the RI that seems pretty easy already (just add the jar)"

Expanding the Use of @Stereotype

A significant majority of 62.3% developers supported expanding the use of @Stereotype across Java EE, only 13.3% opposed. A majority of commenters supported the idea as well as the theme of general CDI/Java EE alignment:

- "Just like defining new types for (compositions of) existing classes, stereotypes can help make software development easier"
- "What about making every Java EE service a CDI extension (in this way, every Java EE annotation would already be handled as a CDI annotation)"
- "But more than that CDI should be expanded to handle remote EJBS, JMS, JMX, etc. Stereotypes are an incredibly powerful and succinct declarative programming tool"
- "Yes, I can see how I would use Stereotypes in existing code to simplify it. I would want the injection/annotation to closely match CDI"
- "Using stereotypes makes using annotations less noisy"
- "The interplay between the old annotations and CDI is confusing enough. The move should be towards CDI and making @Stateless itself a stereotype combining @Transactional and @Pooled"
- "Extending stereotypes is a step in right direction"
- "This is like having inheritance in annotations"
- "Absolutely. This would be awesome. We need this"
- "Just makes basic design principle sense"
- "Definitely, please decouple the EJB component model!"

- "It is so convenient to summarize and abstract one annotation to express a bunch of others. This would reduce 'Annotation Hell' enormously! Please expand the use of Stereotypes to make this possible!"
- "This is especially important if many EJB services are decoupled from the EJB component model and can be applied via individual annotations to Java EE components. @Stateless is a nicely compact annotation. Code will not improve if that will have to be applied in the future as @Transactional, @Pooled, @Secured, @Singlethreaded, @...."
- "This would be a great step forward for consistency and reuse"

Some, however, expressed concerns around increased complexity:

- "Yes, but don't overdo it so it will be the next "xml hell" with 10, 20, 30, 40 annotations grouped together so it will become overwhelming"
- "Could be very convenient, but I'm afraid if it wouldn't make some important class annotations less visible"
- "Could be powerful, but could also be confusing. I'll have to think on this some more..."
- "It would be nice if there were some improved tooling to allow the viewing of all the annotations applied when having @Stereotype annotations applied to another @Stereotype"
- "Terse but not entirely clear for the maintenance developer later"

Expanding Interceptor Use

A very solid 96.3% of developers wanted to expand interceptor use to all Java EE components. 35.7% even wanted to expand interceptors to other Java EE managed classes. Most developers (54.9%) were not sure if there is any place that injection is supported that should not support interceptors. 32.8% thought any place that supports injection should also support interceptors. Only 12.2% were certain that there are places where injection should be supported but not interceptors.

The comments reflected the diversity of opinions, generally supportive of interceptors:

- "It's extremely helpful if interception in the whole EE stack works in one way!"
- "Every EE component should be interceptable and injectable"
- "I would suggest supporting interceptors on all Java EE components to maintain consistency"
- "The whole usage of interceptors still needs to take hold in Java programming, but it is a

powerful technology that needs some time in the Sun. Basically it should become part of Java SE, maybe the next step after lambas?"

- "Yes, make it a complete part of the development model"
- "Might as well make interceptors available everywhere, but provide a guideline for usage"
- "The optimal solution would be to have improved AOP support in Java Standard Edition so that simple interception is possible on all Java classes"
- "If I can @Inject it, I'd like to be able to use a consistent interception approach"
- "It would be handy just to have a blanket approach"
- "I think interceptors are as fundamental as injection and should be available anywhere in the platform"
- "I think that we should have the power to apply interceptors on every architecture slice we want"

A distinct chain of thought separated interceptors from filters and listeners:

- "Servlets already have their 'interceptors' - Filters"
- "I think that the Servlet API already provides a rich set of possibilities to hook yourself into different Servlet container events. I don't find a need to 'pollute' the Servlet model with the Interceptors API"
- "Intercepting anything should be possible at any time. AOP is needed everywhere. That being said, we already can do that for Servlets"