[Experimental] Getting Started with Solr Development

This is a list of key concepts that you should be aware of before doing too much Solr development.

The Rules of Solr

1. The Rules of Solr may be changed by consensus.
2. Community over code.
   We will consider our fellow and future developers before our spiffy code change/fix/feature. We share this project.
3. Develop for long term health.
   We will work towards solutions that can maintain - via enforcement, documentation, good logging, and consideration. I'm not as smart as you, I don't have as much time as you, and I'm not here for the same reasons as you - but there is work I need to do.
4. Code reviews for anything beyond trivial.
   We will not be a kitchen sink or a pet project or test bed because of ill timing or current developer level health.
5. Code to your level.
   We will not commit code we don't understand just because tests pass and will not be reckless in areas we are not qualified yet to be reckless in.

Topics to Familiarize Yourself With.

To be successful with Solr, all you really need is a good attitude. But if you want to go deeper, this is the required reading list:

1. Java Concurrency
2. Good Java Practices
3. Lucene
4. HTTP 1.1 and 2
5. Jetty
6. Testing
7. Gradle
8. Zookeeper
10. Developer Docs

Java Concurrency

This is a very complex topic and I recommend you check out a classic book on the topic.

Object Publishing

A reminder to everyone to please follow the proper rules for object publication - the sharing of an object between two threads. Even if those threads do not access the object concurrently, we must follow these rules. Concurrency is a topic we cannot meet half way.

Proper publication requires that an object's reference AND state are made visible at the same time with the state fully constructed.

A properly constructed object can be shared in one of these ways:

- Initializing the object via a static initializer.
- Using the final keyword on a field to ensure that other threads see a fully constructed object after the constructor returns. The value of the field must be effectively immutable or thread safe. Use Collections.unmodifiable and the like to ensure immutability.
- Using the volatile keyword to ensure that threads read the most up to date value. The volatile keyword can be tricky, but is very cheap when updates are rare.
- Guarding the object via another memory barrier like synchronized or a Lock.

(32bit primitives can be slightly different?)

Class Annotations

Solr uses two annotations to indicate to developers whether a class is thread safe or intended for single threaded use. TODO

Efficiency

We should try and use concurrency strategies that are efficient. For example, see ConcurrentHashMap, ConcurrentHashMap#newKeySet. We should also be careful to size and configure these data structures appropriately. Using a default sized ConcurrentHashMap for a large set of objects is expensive and generally a waste. Forgetting to set the concurrency level appropriately when there is, for example, classloading level concurrency going on, is also a real problem.

We should not do dumb polling - usually this is laziness. Prefer callbacks and #wait #notify #notifyAll or other such efficient mechanisms. If you do poll (don't) for gosh sake, do it at 250ms or above.

Good Java Practices

When we make shortcuts, like violating encapsulation, we create situations that may be reasonable now, but cause problems later. We don't want to be pedantic, but many of the lessons from Object Oriented programming prevent problems down the line.
We also need to pay attention to our class dependency graph - complicated back and forth dependencies can lead to tough code.

We should generally not start or do heavy resource manipulation in an objects constructor. Threads starting up and various interactions before the object is even constructed can be very problematic. Prefer a start() method.

Log things appropriately. Errors that are not logged server side and just go to the client are often painful. Logging a million unnecessary things (every doc update) and then not critical flow steps is often very painful. Adding SolrCore init failures to a tracker that shows up in the UI but not logging the error in the server logs would often be very painful.

Build good scaffolding and then don't destroy it. Take the time to make the tools you use to build with nice and then don't tear them down. Other developers will want and need them too. Don't get creative though - find the projects pulse and stay in synergy or attempt to change the whole flow.

**HTTP 1.1 and 2**

We have to take special care with our use of HTTP as it was not necessarily designed for our use case. Importantly, we don't want to close connections, because we really need and want to reuse them. This means we don't want to close Servlet response streams or use response.sendError calls. Instead we should return a error to the client in the format it asks Solr for as well as the proper response code. We also don't want to flush the response because it will interfere with chunked encoding. We also want to avoid our clients running into stale connections (they hit connection reset exception) - with HTTP 1.1 we can only do this by having the Jetty idle timeout higher the client idle timeout so that the clients control the connection. The other options involving trying to detect a stale connection involve an intrinsic race and are not good enough for our use case.

HTTP2 does not have this race problem and has much hardier connections that multiple requests can be multiplexed over. Solr currently uses a combination of HTTP1.1 and 2.

**Lucene**

[https://lucene.apache.org/](https://lucene.apache.org/)

**Jetty**

Get started here


[https://www.eclipse.org/jetty/documentation/current/optimizing.html](https://www.eclipse.org/jetty/documentation/current/optimizing.html)

**Testing**

[https://labs.carrotsearch.com/randomizedtesting.html](https://labs.carrotsearch.com/randomizedtesting.html)

**Gradle**

**ZooKeeper**

[https://zookeeper.apache.org/](https://zookeeper.apache.org/)

We must take care with our ZooKeeper usage. We want to minimize communication with ZooKeeper. We want to minimize clients to ZooKeeper, watches to ZooKeeper and requests to ZooKeeper. We especially do not want to poll ZooKeeper but instead rely on watchers.

We should use the right types of calls for the right situations:

- **standard request**: We should consider a single request to ZooKeeper expensive in most cases. Prefer to be efficient rather than relying on extra sugar requests we don't need.

- **async api**: This is good for some bulk operations - it's relatively fast because you can send off many payloads without waiting for a response until they are all sent.

- **multi api**: This is known to be the fastest api but also a little trickier with handling responses properly. It also only succeeds if all of the individual operations succeed. You can use a multi to only write data if a certain zknode has a given version as another use case.

We should ensure the ZooKeeper information presented to clients is not inconsistently out of date as much as possible. Waiting to see the collection state change and getting the latest state but very stale live nodes is not a good situation.
Practice defensive coding. Imagine that your change is a game and developers you don't know in 5 years are your adversary. What can you do to ensure they don't ruin your change.

Don't use isolated good practices or singularly configurable timeouts and solutions. Our good practices and timeouts and important things like ConcurrentHashMap sizes and concurrency levels, should all hit a common touch point that we can tweak and allow scaling of easily.