LogonBox Coding Conventions

Language And API

General rules for the source as a whole, and 3rd party frameworks we may use.

All Languages

- All sources to start with copyright header (LogonBox does not yet do automated license headers?)
- **Insert spaces for tabs** OFF
- **Displayed tab width** of 4
- Treat compiler warnings as errors. If you see them, fix them where possible. Or add annotations to ignore them where appropriate.

Java

Java has been on a mission to reduce verbosity. Less code, fewer bugs. So learn and make use of new language features (that are not really new any more!).

- Use Java 11 syntax wherever possible.
- Use `var` keyword everywhere it makes sense. It will save you a lot of typing.
- Make use of `Collection.stream()`. Especially useful for transforming the contents of one collection into another, or filtering lists, or finding distinct objects, or a how of other operations.
- Try to use `Path` and `Files` instead of `File`.
- Use `try-with-resource` instead of closing streams in a `finally`.
- Use the diamond operator `<>` in generics if the type can be inferred another way. E.g. `List<String> l = new ArrayList<>()` rather than `List<String> l = new ArrayList<String>()`.
- Make use of `default` methods in interfaces. This is often preferable to `abstract` classes. Good for event listeners.
- If `null` has meaning to a variables value, consider using `Optional<SomeType>` instead. This can help reduce `NullPointerException`.
- Use Lamba syntax. The most obvious places are with `Runnable` or `Callable` usages and a good way to get used to them.
- Avoid reflection, especially with non-public classes, methods and fields. From Java 16 encapsulation becomes stronger and it is much harder to access these if you are not in control of the modules.
- Use `public`, `private` and `protected` (and lack of) visibility keywords correctly. Only completely omit the prefix if the member is *genuinely* used by other classes in it's package.
- Prefer use of newer date API for new code. (`Instant`, `Period`, `Duration`, `LocalDataTime` etc). See here.

Java Exceptions

When re-throwing and wrapping exceptions, you should nearly always add additional contextual information, and always provide the original as the causal exception.

```
    try {
       // some code
    } catch(ResourceException re) {
       throw new IOException("A bad thing happened.", re);
    }
```

Try to use multi-catch ...

```
 try {
    // some code
}
catch(ResourceException | IOException e) {
    // handle generically
}
```

Don't wrap exceptions if you can-rethrow a super type.

```
 try {
    // some code
}
catch(FileNotFoundException fe) {
    // special handling
    throw fe;
}
catch(IOException ioe) {
    // generic io exception handling
    throw ioe;
}
catch(Exception e) {
    // all other
    throw new IOException("A bad thing happened.", e);
}
```

Always log exceptions to the logging framework, don't just dump them to System.out.

```
 try {
    // some code
}
catch(IOException ioe) {
    LOG.error("A bad thing happened.", ioe); // do this
    //ioe.printStackTrace(); // don't do this
}
```

Java Source Format

Up for discussion, but I would like to say use the Eclipse default. If this is not acceptable to others, we can decide on a format and create a formatter profile for it.

I believe there is a way of importing of these can be stored in a source tree and automated.

Logging

- Wherever possible do not use string concatenation or `String.format()` in log statements. The logging framework now in use supports patterns. So instead use `LOG.error("A bad thing happened to the user '{}'.", username);`
- The static variable name of the `Logger` instance should be capitalised, it's a constant. It should also nearly always be `private` and `final`.
- Always use `LOG.isDebugEnabled()` for debug logging (and lower levels). Use discretion for higher log levels based on performance impact (i.e. how much processing time does it take to build the logging string), and the context in which it is used (is it a performance sensitive area).

Comments

Source code comments can be useful, but also can become inaccurate or irrelevant over time. They can get detached from the code they apply as refactoring happens. They may be ugly and detract from the code itself.

To keep this to a minimum, consider ..

- Is the comment of value to other developers? If it being used as an aid during development, remove them before committing.
- Is the comment describing a piece of hard to understand code? Consider refactoring so it is readable (more methods, more descriptive variable names).
- Is the comment a bug? Perhaps fix the bug!

If you must comment, use the following format. Eclipse supports "Task Tags". There are 3 of these enabled by default, so make use of them. You can then use the *Tasks* view to see them in the IDE.

- `FIXME` . A bug. Prefer only used when caused by external influences such as 3rd party libraries, or added during code-review. I.e, do not commit NEW bugs.
- `TODO` . Generic task, prefer these to only be used during development. It can understable for these to exist if development is in stages, but a project should not be considered completed until the TODOs are done or removed. Ask yourself if a ticket should be created as a reminder if the task is substantial.
- `XXX` . Everything else you want to appear in tasks list. Use sparingly.
- `INFO` . For all other comments. These will not appear in the Eclipse task list unless you add the tag in preferences.

Also, initial and date the comment.

```
/* FIXME BPS - 30/07/22. This is a bug, put a title here.
 * And this is the full description. Use as many lines
 * as you like or need.
 */
```

If you refer to other Java classes in a comment, remember to use double asterisks to open and close the comment. This makes Eclipse highlighting and links work correctly.

```
/** INFO BPS - 30/07/22. This piece of code is complex.
 * For a description of the algorithm used here, see
 * the Javadoc for {@link MyComplexInterface}.
 */
```

JavaScript

- Prefer single quotes for strings

LogonBox Framework

Some guidelines on using the LogonBox/Hypersocket framework specifically.

- Take care to not `@Autowire` stuff you don't need, it has a cost. Never leave as package protected so IDE has a change of noticing.
- Do not use `Collection<SomeResource> SomeResourceService.allResources()` methods in service layers, unless you are absolutely *sure* the table will *always* be small. Prefer `Iterator<SomeResource> iterate()` methods.

Projects

- If you are starting a new Java project (a library or an application), strongly consider writing it using Java modules (JPMS). If a dependency that you wish to use doesn't have it (at least an `Automatic-Module-Name`), consider finding an alternative. At this stage the lack of module information in a library suggests lack of maintenance, or technical debt that cannot be overcome (such as heavy use of reflection or other less than ideal techniques being used).
- If you do use JPMS, make use of public and private packages to hide implementation code from consumers where

appropriate.

- Ensure all packages within a module are unique across the application. If you are writing a public module, it should be *globally unique*.