qooxdoo 3 Part Handling

Thomas Herchenröder
1&1 Internet AG

Parts

Parts are a means to *logically* partition a qooxdoo application, so that those parts can be loaded *incrementally* and *on demand*. Parts are defined by the user through configuration entries. The *Generator* uses this configuration to distribute class code and resource information across multiple script files that are then retrieved via HTTP. The aim is to avoid loading of unnecessary code into the browser.

Concepts

In order to package the classes of an application (including classes of the framework, contribs and any additional libraries) in a way that makes for efficient loading, the Generator collects class code into *scripts* (.js files). Scripts are grouped into *packages*. Scripts of the same package are always loaded at the same time (if with multiple HTTP requests). Each *part* is implemented by a collection of packages, and a package might be required by multiple parts. qooxdoo's *PartLoader* loads all packages for a part that has been *require*'d, and have not been loaded yet.

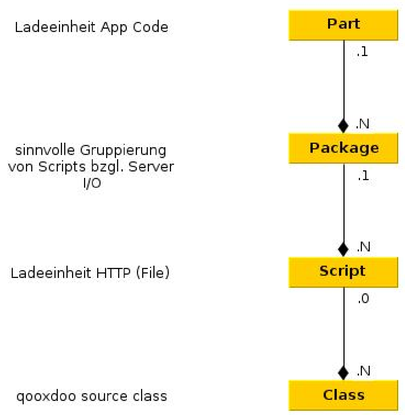

Figure 1: Relations between parts, packages, scripts and classes.

2-Phase Package Calculation

The essence of the realization of parts is the calculation which classes should belong to which package. Everything else is derived from that. The package calculation process is devided in two steps:

- partition the set of code classes into **equivalence sets**
  This means grouping all classes together that are *required by the same set of parts*. These groups will be the initial packages. There will usually be a large number of them, some with only a few classes.
- **collapsing** (or merging) those packages together, so that bigger packages can be fetched with fewer HTTP requests, while still preserving that no unnecessary code is loaded into the browser.
  This will significantly reduce the total number of packages.

Equivalence sets

To come up with the equivalence sets for the classes,

- calculate the classes necessary for each part, starting from the part's *"include"* config.
- for each class in the built, collect the set of parts which require this class.
- group classes that are required by the same set of parts.

For example, if you have defined parts "One" and "Two", there will be (at most) tree equivalence sets: One set of classes that are required by exactly part "One", one set of classes that are required by exactly part "Two", and one set of classes that are required by both ["One", "Two"].
The maximum number of equivalence classes can be expressed as the binomail coefficient $\binom{N}{k}$, but for a particular built there might be less. (TODO).

Conditions for Package Collapsing

When collapsing two packages, one is the receiving package to which the classes of the second are added, and the other which yields its classes and is then removed from the list of packages. For this to succeed, a number of constraints have to be observed, or the package calculation will result in faulty packages (i.e. packages which will not load cleanly in the browser e.g. because of unresolved symbols).

- the receiving package must be loaded for *at least* the same parts as the yielding package (i.e. the equivalence class of the yielding package must be a subset of that of the receiving package)
- all load-time dependencies of each class of a package must be *within* this package (in load order), or must be in packages that are loaded earlier at runtime (TODO)
- in all parts where the second package would have been loaded, now the first package must be loaded in its place (this might result in unnecessary classes being loaded for that part, which is a trade-off)

References

Some references and a graphic to show you how it's done:

[1] D. W. Kribs, R. Laflamme, D. Poulin, M. Lesosky, Quantum Inf. & Comp. **6** (2006), 383-399.

[2] P. Zanardi, M. Rasetti, Phys. Rev. Lett. **79**, 3306 (1997).