Interactive Session 2 – Search and basic composition

Search and Composition

In this session we explore how to search for components, and how to write the beginnings of your own system composition service.

We start by writing a simple composer which uses the `composition.Search` API to find other components.

In the same folder as your project from the last session, create a new file called `MyCom.dn` and enter the usual entry-point code:

```
component provides App requires io.Output out, composition.Search search {

        int App:main(AppParam params[])
                {


                return 0
                }
}
```

We're now going to read the first command-line parameter and assume that this is itself a path to another entry-point component of the system we're going to compose. We can examine the dependencies of this component using the `composition.ObjectWriter` API, to then determine which other components we need to find in order to form a working system.

The ObjectWriter is a component which can examine the meta-data attached to a compiled component and return requested parts of that meta-data. This meta-data is extensible and can use any syntax you like; the ObjectWriter can also inject new meta-data into existing components. The required and provided interfaces of each component are automatically added as a meta-data field by the Dana compiler, using JSON as the encoding format. Reading this list of interfaces therefore involves both the ObjectWriter to extract the raw meta-data and a JSON decoder to extract the content.

We've written a helper function below which does all of this for you on the following page. It extracts the special meta-data section"DNIL" (Dana Interface List) from the component object file, then passes this data into a JSON parser, and finally uses the parser to walk through the JSON document to extract the list of required interfaces as an array.

```
data ReqIntf {
     char package[]
     char alias[]
     bool isNative
     }

component provides App requires io.Output out, composition.Search search, data.json.JSONParser parser,
composition.ObjectWriter {

     ReqIntf[] getRequiredInterfaces(char com[])
        {
        ReqIntf result[]

        ObjectWriter reader = new ObjectWriter(com)
        InfoSection section = reader.getInfoSection("DNIL", "json")

        JSONElement document = parser.parseDocument(section.content)

        JSONElement requiredInterfaces = parser.getValue(document, "requiredInterfaces")

        if (requiredInterfaces != null)
            {
            for (int i = 0; i < requiredInterfaces.children.arrayLength; i++)
               {
               JSONElement ri = requiredInterfaces.children[i]
               char package[] = parser.getValue(ri, "package").value
               char alias[] = parser.getValue(ri, "alias").value
               bool isNative = parser.getValue(ri, "native") != null && parser.getValue(ri, "native").value == "true"

               result = new ReqIntf[](result, new ReqIntf(package, alias, isNative))
               }
            }

        return result
        }

     int App:main(AppParam params[])
        {
        char mainPath[] = params[0].string

        ReqIntf rql[] = getRequiredInterfaces(mainPath)

        for (int i = 0; i < rql.arrayLength; i++)
            {
            out.println("required interface: $(rql[i].package)")
            }

        return 0
        }

     }
```


If you update your MyCom.dn component with this code, then compile it:

dnc MyCom.dn

You can then run this against your existing example program using the command:

dana MyCom.o Main.o

Here you should see a list of required interfaces printed out to the console.

For each of these required interfaces, we can then use the standard search approach (using our default naming convention) to acquire a list of known components that implement them.

As an example, this can be done by updating the "main" function to be:

```
int App:main(AppParam params[])
    {
    char mainPath[] = params[0].string

    ReqIntf rql[] = getRequiredInterfaces(mainPath)

    for (int i = 0; i < rql.arrayLength; i++)
        {
        out.println("required interface: $(rql[i].package)")

        String lst[] = search.getComponents(rql[i].package)

        for (int j = 0; j < lst.arrayLength; j++)
            {
            out.println(" -- available component: $(lst[j].string)")
            }
        }

    return 0
    }
```

We now have a list of components which could potentially be used to satisfy each dependency of our target component. We can then query each of these components to examine their required interfaces, and find a set of possible components that can be used to satisfy those, and so on.

Having done this, we can then use the `dana.rewire()` command used in the previous session to wire up a working system. Unlike previous session, however, our composer is now *generalised* to any system rather than having any hard-coded components.

The rest of this session is left to allow you to finish this composer for yourself to enable it to build a fully working system.

Aside from the composition algorithm itself (which we leave to you), there are two extra factors to consider when doing this, noted in the appendix.

Appendix

1. Some interfaces are automatically linked against the Dana runtime, and so don't have a "component". These interfaces don't need to be wired up by you.

You can detect which interfaces have this property as follows: first, add a required interface of type System:

```
component provides App requires io.Output out, composition.Search search, data.json.JSONParser
parser, composition.ObjectWriter, System system {


    }
```

And then write a function which uses this API to check if a particular interface is "automatically bound":

```
    bool isAutoBinding(char package[])
       {
       String intfs[] = system.getAutoBindings()

       for (int i = 0; i < intfs.arrayLength; i++)
           {
           if (intfs[i].string == package) return true
           }

       return false
       }
```

2. Some interfaces are linked against native libraries (which are written in C). Instead of using the regular Loader interface (which you used in the previous session, and which is used to load all regular components), to load these native libraries, you need to use the NativeLoader.

```
component provides App requires io.Output out, composition.Search search, data.json.JSONParser
parser, composition.ObjectWriter, System system, NativeLoader loader {


    }
```

If you know that the component you're loading is a native library (identified by the isNative flag in ReqIntf), you can then use the following code to load it (and wiring is done as normal):

```
            IDC ncom = nLoader.load(ri[i].alias)
            dana.rewire(newCom.com :> ri[i].alias, ncom :< ri[i].alias)
```