3.x Spring Proposal

Bean Definition Loading

1. The Server impl creates a root context, and registers itself as a singleton. It creates a child context for application bean definitions, and loads a definition for ServerConfiguration as a bean.

2. It creates a definition for ModelBasedTripleGenerator that can be over-ridden.

3. It loads all xml definitions in FEDORA_HOME/server/config/spring

4. It parses the fcfg, creating definitions for Modules, ModuleConfigurations, and DatastoreConfigurations

5. It loads any definitions created in loadBeanDefinitions

6. It refreshes the child context.

Module Creation

The base Server class is refactored to use a Spring application context to resolve Module references. Bean definitions for Modules are built up programmatically based on the existing, reflection-based Module instantiation and parsed fcfg data. The Server is auto-wired into the modules. Module configurations and Datastore configurations are loaded as beans as well, with the former named as role + "Configuration", and the latter named with the datastore id. This allows existing code that does not need access to Spring beans to run unchanged.

```
    protected static GenericBeanDefinition createModuleBeanDefinition(String className, Object params, String
role){

       GenericBeanDefinition result = new GenericBeanDefinition();
       result.setScope(BeanDefinition.SCOPE_SINGLETON);
       result.setBeanClassName(className);
       result.setAttribute("id", role);
       result.setAttribute("name", role);
       result.setAttribute("init-method", "initModule");
       result.setAttribute("destroy-method", "shutdownModule");
       result.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_CONSTRUCTOR);
       result.setParentName(Module.class.getName());

       ConstructorArgumentValues cArgs = new ConstructorArgumentValues();
       cArgs.addIndexedArgumentValue(0, params,MODULE_CONSTRUCTOR_PARAM1_CLASS);
       BeanReference serverRef = new RuntimeBeanReference(MODULE_CONSTRUCTOR_PARAM2_CLASS);
       cArgs.addIndexedArgumentValue(1, serverRef);
       cArgs.addIndexedArgumentValue(2, role,MODULE_CONSTRUCTOR_PARAM3_CLASS);
       result.setConstructorArgumentValues(cArgs);
       return result;
   }



    protected static GenericBeanDefinition createModuleConfigurationBeanDefinition(String role){
       GenericBeanDefinition result = new GenericBeanDefinition();
       result.setScope(BeanDefinition.SCOPE_SINGLETON);
       result.setBeanClassName(ModuleConfiguration.class.getName());
       String name = role+"Configuration";
       result.setAttribute("id", name);
       result.setAttribute("name", name);
       result.setFactoryBeanName(ServerConfiguration.class.getName());
       result.setFactoryMethodName("getModuleConfiguration");

       ConstructorArgumentValues cArgs = new ConstructorArgumentValues();
       cArgs.addGenericArgumentValue(role);
       result.setConstructorArgumentValues(cArgs);
       return result;
   }



    protected static GenericBeanDefinition createDatastoreConfigurationBeanDefinition(String id){

       GenericBeanDefinition result = new GenericBeanDefinition();
       result.setScope(BeanDefinition.SCOPE_SINGLETON);
       result.setBeanClassName(DatastoreConfiguration.class.getName());
       result.setAttribute("id", id);
       result.setAttribute("name", id);
       result.setFactoryBeanName(ServerConfiguration.class.getName());
       result.setFactoryMethodName("getDatastoreConfiguration");

       ConstructorArgumentValues cArgs = new ConstructorArgumentValues();
       cArgs.addGenericArgumentValue(id);


       result.setConstructorArgumentValues(cArgs);
       return result;

    }
```

The existing getModule method becomes a proxy for the new getBean(String name, Class<? extends Module> type)

Non-Module Creation

Server impls can override a newly added method to load custom Bean definitions. For example, RebuildServer loads definitions of the default Rebuilder implementations. The trippi TriplestoreConnector class and the ModelBasedTripleGenerator class have some plain java refactorings (addition of setters and getters) to support extension and/or override of existing functionality (See also FCREPO-743). These bean definitions can be loaded from the Spring config directory, or can just be scanned classes:

```
    @Override
    protected void registerBeanDefinitions()
        throws ServerInitializationException{
        for (String rebuilder:REBUILDERS){
            try{
                ScannedGenericBeanDefinition beanDefinition =
                    Server.getScannedBeanDefinition(rebuilder);
                beanDefinition.setAutowireCandidate(true);
                beanDefinition.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_BY_NAME);
                beanDefinition.setAttribute("id", rebuilder);
                beanDefinition.setAttribute("name", rebuilder);
                beanDefinition.setScope(BeanDefinition.SCOPE_SINGLETON);
                registerBeanDefinition(rebuilder, beanDefinition);
            }
            catch (IOException e){
                throw new ServerInitializationException(e.toString(),e);
            }
        }
    }
```

Non-module beans can be refactored to avoid reliance on the Server impl to locate information:

```
    @Resource(name = "org.trippi.TriplestoreConnector")
    public void setTriplestoreConnector(TriplestoreConnector conn){
        m_conn = conn;
    }

    @Resource(name = configName)
    public void setModuleConfiguration(ModuleConfiguration riConfig) {
        m_riConfig = riConfig;
    }
```

... Or they can be Spring-aware, and get Beans directly:

```
     private static Rebuilder getRebuilder() throws Exception {private static Rebuilder getRebuilder() throws
Exception {
        Server server = getServer();
        String [] rebuilders = server.getBeanNamesForType(Rebuilder.class);
        String[] labels = new String[rebuilders.length + 1];
        int i = 0;
        for (i = 0; i < rebuilders.length; i++) {
            Rebuilder r =
                server.getBean(rebuilders[i],Rebuilder.class);
            labels[i] = r.getAction();
        }
        labels[i] = "Exit";
        int choiceNum = getChoice("What do you want to do?", labels);
        if (choiceNum == i) {
            return null;
        } else {
            return server.getBean(rebuilders[choiceNum],Rebuilder.class);
        }
    }
        Server server = getServer();
```