Introduction into FT

Oliver Langer / Version 0.1.0

Table of Content

1. TODO

- Proofreading of this document, removing mistakes
- Write a “real” documentation

2. About this document

This is a very short introduction into FT. In future this should be replaced by a "real" documentation. At present you may take a look into the testcases under "tests/" in order to see possible applications.

The following steps show how to create and modify a graph.

3. Create a Runtime Environment

Create a directory `Sample` where a sample project may be placed in. With respect to your environment (OS X or GNUstep) proceed as follows...

Under OS X

For OS X simply create a new Foundation Tool project within `Sample` and add the frameworks Encore, BDB and FT to it.

Under GNUstep

Create a GNUmakefile within `Sample` which looks as follows:

```
include $(GNUSTEP_MAKEFILES)/common.make

DBROOT=/home/ola/installed

TOOL_NAME=FTSample
VERSION=0.1

ADDITIONAL_INCLUDE_DIRS=\
  -I$(DBROOT)/include

ADDITIONAL_NATIVE_LIBS += BDB Encore FT

FTSample_OBJC_FILES=\
  FTSample.m

FTSample_HEADER_FILES=

-include GNUmakefile.preamble
include $(GNUSTEP_MAKEFILES)/tool.make
-include GNUmakefile.postamble
```

Adjust the parameter `DBROOT` so that it points to the root of your berkeley db installation.

4. Create FTSample.m

Create the source `FTSample.m` within the directory `Sample`. The file should initially contain the following lines of code:

```
#include <Foundation/Foundation.h>

int main( int argc, char *argv[] ) {
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

  NSLog( @"**FTSample " );

  [pool release];

  return 0;
}
```

Next compile the program via calling “make” and then start it. In the following sections `FTSample.m` will be enhanced step by step.

5. Starting the Server

For the startup, the server expects some configuration information. Such information includes e. g. the root location of the data files which are used for internal managent data. So before starting the server, the configuration has to be set up. This is done within an XML file.

Configuration of the Server

The following xml file may be stored as `ftconfig.xml` and resembles a suitable configuration for the sample:

```
<FTConfig
  baseDataDir="./ftdata">
  <DatabaseNames class="FTConfigDatabaseNames">
    <DatabaseName class="FTConfigDatabaseName"
       entry="FTDatabaseName.DefaultObject2IdMapper"
       databaseName="defaultObject2Id.db"/>

    <DatabaseName class="FTConfigDatabaseName"
       entry="FTDatabaseName.DefaultDictionaryProvider"
       databaseName="defaultDictionaryProvider.db"/>

    <DatabaseName class="FTConfigDatabaseName"
      entry="FTDatabaseName.NodeId2ProviderManager"
      databaseName="node2provider.db"/>

    <DatabaseName class="FTConfigDatabaseName"
      entry="FTDatabaseName.graphid2Graph"
      databaseName="graphId2graph.db"/>

    <DatabaseName class="FTConfigDatabaseName"
      entry="FTDatabaseName.graphDBNameScheme"
      databaseName="graph-%d.db"/>

    <DatabaseName class="FTConfigDatabaseName"
      entry="FTDatabaseName.objectToIdMapper"
      databaseName="objectToId.db"/>

    <DatabaseName class="FTConfigDatabaseName"
```

```
      entry="FTDatabaseName.nodes"
      databaseName="nodes.db"/>

    <DatabaseName class="FTConfigDatabaseName"
      entry="FTDatabaseName.id2Recno"
      databaseName="idToRecno.db"/>
  </DatabaseNames>

  <Services class="FTConfigServices">
    <ServiceLoaders class="FTConfigServiceLoaders">
      <ServiceLoader
        class="FTDictionaryServiceLoader"/>
    </ServiceLoaders>
  </Services>
</FTConfig>
```

Most of the entries of this file specify the name of data files and may be left as given. There are only two entries of interest; both entries are marked bold in the above listing:

- `baseDataDir`: Specifies the directory or subdirectory under which the server may store all data. It must be possible for the server to create files and subdirectories within the given directory.
- `ServiceLoader`: The service loader section enables the addition of services through the registry of service loaders. At present FT only supports a simple dictionary service allowing a client to store and retrieve blobs through keys.

The `baseDataDir` refers to "`./ftdata`". Therefor the corresponding subdirectory `ftdata` has to be created within `Sample`.

The resulting XML file `ftconfig.xml` has to be hand-off to the server via the process parameter `-configFile`. In our example this means that FTSample has to be called in this way: `FTSample -configFile ftconfig.xml`.

Code for Starting the Server

To start the server an instance of `FTBootstrap` has to be obtained. By obtaining this instance, the server reads and interpretes the configuration file (`ftconfig.xml`). Afterwards the server is ready for startup. Than, starting the server results in setting up data files for internal usage which makes the server ready for usage.

The resulting code additions used to start the server make `FTSample.m` look as follows:

```
#include <Foundation/Foundation.h>
#include <FT/FT.h>


id <FTServer> server = nil;

void startServer( BOOL firstStartOfServer ) {
  FTBootstrap *bootstrap;

  bootstrap = [FTBootstrap instance];

  if( firstStartOfServer ) {
    server = [[bootstrap initializeServer] retain];
  } else {
    server = [[bootstrap startServer] retain];
  }
  [bootstrap release];
}


int main( int argc, char *argv[] ) {
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  FTBootstrap *bootstrap;

  NSLog( @"**FTSample " );

  startServer(YES);

  [pool release];

  return 0;
}
```

The two lines marked bold demonstrate the initialization and start of a server. The initialization only takes part for the first time a server is started. In this case the server creates diverse internal data files. All subsequent starts of the server may use `startServer`.

6. Getting a session

With a valid server instance, getting a session functions in the following order: You fetch a so-called session manager from the server. With this manager you can create a session for “a particular login” - at present FT only supports one login (login=”`altum`”, password=”`silentium`”) which results in having an administrative session.

The code to be inserted into `FTSample.m`:

```
id <FTSession> login(
  NSString *loginId, NSString *passwd )  {
  id <FTSessionManager> sessionManager;
  id <FTSession> toReturn;

  sessionManager = [server sessionManager];
  toReturn = [sessionManager loginAs: loginId withPassword: passwd];

  return [toReturn autorelease];
}
```

The related modifications of the `main()`-methods are straight forward:

```
int main( int argc, char *argv[] ) {
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  FTBootstrap *bootstrap;
  id <FTAdministrationSession> session;

  NSLog( @"**FTSample " );

  startServer(YES);
  session = (id <FTAdministrationSession>) login( @"altum",
@"silentium" );
```

```
  [pool release];

  return 0;
}
```

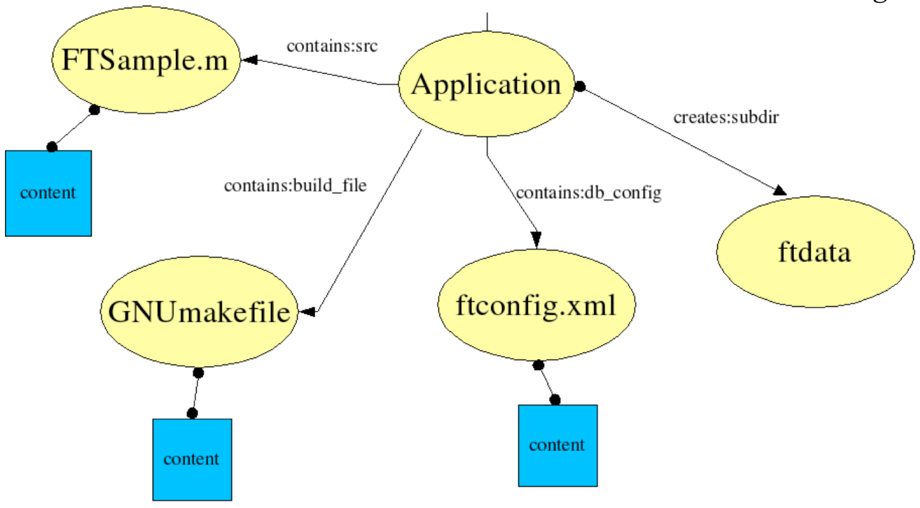
7.Creation of a Graph

Before we create a graph we should introduce a small domain we would like to model and manage with FT. Afterwards we have to introduce some basics regarding the datamodel. Then we are prepared to create our first graph.

A Simple Application Domain

In the example we use FT to manage all files of `FTSample` in a very simple repository. That is we

- store all files used to build and run `FTSample` in FT and
- store information about all directories we have to create before running `FTSample`.



Managing this information we could potentially “checkout” and “set up” the project at a different location. For the storage of the data we need a model based on the artefacts FT supports. The following figure illustrates a possible model:

In terms of the data model of FT, the figure consists of nodes (circles), edges (lines) and dictionary services (rectangles). In the displayed model, the node `Application` is the central point of interests. The outgoing edges with prefix “`contains`” refer to nodes which:

- stand for files and which
- store the content the files.

The edge with prefix “`creates`” refers to a node where

- the related directory has to be created.

So when we create a graph, we need to read in all necessary files used to compile `FTSample` and have to create corresponding nodes. Each node has to be filled with the content of the underlying file and has to be linked to `Application` with an edge with prefix “`contains`”. Afterwards we create nodes for all resources which have to be created in order to correctly compile/run `FTSample.`

Proceeding this way we should have stored all required data of the example in FT. Thus we should be able to "check out" the example at any location within a file system. During such a "checkout" all necessary files have to be created.

To make things not to complicated, we do not put any more functionality here.

Some Basics of the Datamodel

Before we step into the creation of the corresponding graph we may take a short look into important characteristics of the datamodel of FT.

Identifiers in FT

FT works with an internal format used to represent identifiers. All artefacts you create/use with FT rely on such identifiers. Objects implementing the protocol `FTObjectToIdMapper` help you creating identifiers. `FTObjectToIdMapper` offers methods to map an given object to a related FT identifier which implements the protocol `FTId`. At present, `FTObjectToIdMapper` only supports the mapping of instances of `NSString`.

What does this mapping mean? Imagine you would like to create a graph with name "HelloWorld". To do so you have to hand off the string "HelloWorld" to a given `FTObjectToIdMapper` instance in order to map it to a `FTId` instance. Now this instance of `FTId` can be used to create the graph.

There is only one additional thing to know about the handling of identifiers:

1. Identifiers related to graphs are managed by a global instance of `FTObjectToIdMapper` which can be obtained through `[myFTSession defaultObjectToIdMapper]`.
2. Identifiers related to elements of a graph may only be obtained by instances of `FTObjectToIdMapper`, which are managed and given by the underlying graph. For this purpose, each graph instance supports the method `[myGraph objectToIdMapper]`.

Transactions in FT

Although the support of transactions in FT is quite poor, all modifications within FT must be enclosed in a transaction context. At present, FT does not (really support transactions anyway...) nested transactions and also it does not support any special setting. So each of your operation sections may start with the following code sniplet:

```
id <FTTransaction> transaction = nil;
...
transaction = [session beginTransactionWithParent: nil withSettings:
nil];
```

and analogeously end with

```
[transaction commit];
```

Nodes

Nodes are created, loaded, modified or destroyed through graph instances. A node consists of

- a node identifier,

- a set of outgoing edges,
- a set of incoming edges and
- a set of services attached to it.

The current interface for node instances offers these methods:

```
*!
 * @protocol FTNode
 * @abstract Interface of all node instances within FT
 */
@protocol FTNode <FTObject>

/*!
 * @method createAndAppendEdgeWithId
 * @abstract creates an edge which refers to the given node
 * @param edgeId id of the edge
 * @param targetNode node to which the edge refers
 * @result created edge instance
 */
- (id <FTEdge>) createAndAppendEdgeWithId: (id <FTId>) edgeId
  withTargetNode: (id <FTNode>) targetNode;


/*!
 * @method nodeId
 * @result return the id of this node.
 */
- (id <FTId>) nodeId;


/*!
 * @method outgoingNodes
 * @abstract access to all nodes referred by the receiver of this
message
 * @result iterator over instances which implement the protocol FTNode
 */
- (id <ECIterator>) outgoingNodes;


/*!
 * @method removeAllOutgoingNodeWithId
 * @abstract removes all outgoing nodes with the specified id
 * @param nodeId identifier for the node
 * @result self
 * @throws ECIllegalArgumentException if the node with the given id
could
 *   not be found
 */
- removeAllOutgoingNodesWithId: (id <FTId>) nodeId;


/*!
 * @method serviceWithId
 * @abstract retrieve the specified service
 * @param aServiceId identifier for the service
 * @result related service or nil, if not found or version cannot be
found
 */
- (id <FTService>) serviceWithId: (NSString *) aServiceId;
@end
```

The sets of outgoing and incoming edges are ordered. But at present the user has no operations to modify any order. One constraint seems to be hindering when using edges: A set of edges (incoming or outgoing) may only contain edges with distinguish names. This constrain is likely to be changed in future!

Please remind that with respect to the current state of the implementation there are quite a lot of methods missing.

Edges

Edges are used to link nodes to each other. Edges consist of

- an edge id,
- a source node and
- a target node

Edges are unidirectional. Two nodes may not be linked twice with the same edge or with two edges with the same name. At present edges to not store any additional information (cardinality, services etc.).

Services

Services are attached to particular or any nodes and offer functions which may be related to the particular node they are attached to. At present only a dictionary service is provided which offers the storage of multiple blobs per node.

Each service is identified through a service id. All services only have a very small protocol in common, allowing to ask for the service id and for the services' version. It is not planned to introduce common protocols for “common” interfaces. This means that the interface of service is always particular to that service. The protocol for the dictionary provided hereby looks as follows:

```
/*!
 * @class FTDictionaryService
 * @abstract A service which allows usage of simple dictionaries for
nodes.
 * @discussion This service returns "FTDictionaryService" as service id
and
 * allows simple management of key/value pairs.
 */
@protocol FTDictionaryService <FTService>

/*!
 * @method allKeys
 * @result all keys
 */
- (id <ECIterator>) allKeys;


/*!
 * @method close
 * @abstract closes this dictionary. After this call the dictionary may
not
 *   be used any more
 */
- (void) close;


/*!
 * @method setObject
 * @abstract adds an object to the dictionary. Replaces an existing one
if there
 *   is one already defined for the given key
 * @param anObject object to add
 * @param aKey underlying key
 * @result self
 */
- setObject: (id <NSCoding>) anObject forKey: (id <NSCoding>) aKey;


/*!
 * @method objectForKey
 * @abstract returns the object for the given key
 * @param aKey underlying key
 * @result object for specified key
 */
- objectForKey: (id) aKey;
@end
```

Graphs

Graphs are created or removed through a graph manager, which can be obtained by the session. A graph consists of

- a graph id,
- a set of nodes
- a set of services and
- a mapping object, which maps objects to identifiers.

Graphs do not have any dependency and so they e.g. cannot share any resources (nodes, egdes etc.). In other words, all elements created within a graph “belong” to that graph. Also identifiers have to be defined locally wrt. a graph. For this purpose each graph offers its own mapping object. Clients should always use this mapping object and never use the systems' default object mapper.

The current set of supported methods:

```
/*!
 * @protocol FTGraph
 * @abstract Instances of this protocol represent graphs
 */
@protocol FTGraph <FTObject>

/*!
 * @method createNodeWithId
 * @abstract create a node of a certain node id
 */
- (id <FTNode>) createNodeWithId: (id <FTId>) aNodeId;


/*!
 * @method graphId
 * @result the identifier of this graph
 */
- (id <FTId>) graphId;


/*!
 * @method close
 * @abstract Call this method if you do not need this instance any more
 * @discussion After this method release may also be called
 */
- (void) close;


/*!
 * @method nodeWithId
 * @result return the node corresponding to the given identifier
 */
- (id <FTNode>) nodeWithId: (id <FTId>) aNodeId;


/*!
 * @method objectToIdMapper
 * @result mapper for this graph
 */
- (id <FTObjectToIdMapper>) objectToIdMapper;


/*!
* @method serviceWithId
 * @abstract Fetch a service for the specified id
 * @param aServiceId id of the service
 * @result the specified service or nil if not existent
 */
- (id <FTService>) serviceWithId: (NSString *) aServiceId;
@end
```

Modelling the Application Domain in FT

Regarding the example application domain we have to do the following in terms of the shortly introduced datamodell:

I. Create an identifier for a new graph using the systems' default object mapper.

II. Create a graph using the previously fetched identifier .

III. Obtain the object mapper of the created graph.

IV. Create five identifiers mapped from `FTSample.m`,`GNUmakefile`, `Application`, `ftconfig.xml`, `ftdata` using `om`.

V. Create five nodes with identifiers.

VI. Create four identifiers used to create the edges.

VII. Create the four edges using `previously created identifiers`.

VIII. Now store the content of the files `FTSample.m`, `GNUmakefile`, `ftconfig.xml` using the service FTDictionaryService of the related node. Looking at the resulting code you will examine some redundancy. But to keep the example simple we do not add any utility functions and the like.

IX. Now commit the changes.

With these steps in mind, the code may look like to be found in the function `createGraph ()`:

```
void createGraph( id session ) {
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  id <FTTransaction> transaction;
  id <FTObjectToIdMapper> sysOM;
  id <FTGraphManager> graphManager;

  id <FTObjectToIdMapper> graphOM;
  id <FTId> graphId;
  id <FTGraph> graph;
  id <FTId>
id_application,id_ftsample,id_buildFile,id_ftconfig,id_ftdata;
  id <FTNode>
node_application,node_ftsample,node_buildFile,node_ftconfig,node_ftdata
;
  id <FTId> containsSrcId, containsBuildFileId, containsDbConfigFileId,
    createsSubDirId;
  id <FTDictionaryService> dictionaryService;
  NSData *fileContent;

  transaction = [session beginTransactionWithParent: nil withSettings:
nil];

  sysOM = [session defaultObjectToIdMapper];
  graphManager = [session graphManager];

  // start transaction

  // Step I: create identifier for graph
  graphId = [sysOM mapObject: @"FTSampleGraph"];

  // Step II: create graph
  graph = [graphManager createGraphWithId: graphId];

  // Step III: get object mapper of the created graph
  graphOM = [graph objectToIdMapper];

  // Step IV: create the five identifiers:
  id_application = [graphOM mapObject: @"Application"];
  id_ftsample = [graphOM mapObject: @"FTSample.m"];
  id_buildFile = [graphOM mapObject: @"GNUmakefile"];
  id_ftconfig = [graphOM mapObject: @"ftconfig.xml"];
  id_ftdata = [graphOM mapObject: @"ftdata"];

  // Step V: create the five nodes
  node_application = [graph createNodeWithId: id_application];
  node_ftsample = [graph createNodeWithId: id_ftsample];
  node_buildFile = [graph createNodeWithId: id_buildFile];
  node_ftconfig = [graph createNodeWithId: id_ftconfig];
  node_ftdata = [graph createNodeWithId: id_ftdata];

  // Step VI: create all edge identifiers:
  containsSrcId = [graphOM mapObject: @"contains:src"];
  containsBuildFileId = [graphOM mapObject: @"contains:buildFile"];
  containsDbConfigFileId = [graphOM mapObject:
@"contains:dbConfigFile"];
  createsSubDirId = [graphOM mapObject: @"creates:subDir"];

  // Step VII: Link the nodes appropriately to node node_aplication:
  [node_application createAndAppendEdgeWithId: containsSrcId
    withTargetNode: node_ftsample];

  [node_application createAndAppendEdgeWithId: containsBuildFileId
    withTargetNode: node_buildFile];

  [node_application createAndAppendEdgeWithId: containsDbConfigFileId
    withTargetNode: node_ftconfig];

  [node_application createAndAppendEdgeWithId: createsSubDirId
    withTargetNode: node_ftdata];

  // Step VIII
  // now store the content of the file FTSample.m within the node
  // "node_ftsample" using the dictionary service:

  // store FTSample.m in node_ftsample:
  dictionaryService = (id <FTDictionaryService>)
    [node_ftsample serviceWithId: @"FTDictionaryService"];

  fileContent = [[[NSData alloc] initWithContentsOfFile: @"FTSample.m"]
    autorelease];
  assert( nil != fileContent );

  [dictionaryService setObject: fileContent forKey: @"fileContent"];

  [dictionaryService close];
  // store GNUmakefile in node_buildFile
  dictionaryService = (id <FTDictionaryService>)
    [node_buildFile serviceWithId: @"FTDictionaryService"];

  fileContent = [[[NSData alloc] initWithContentsOfFile:
@"GNUmakefile"]
    autorelease];
  assert( nil != fileContent );

  [dictionaryService setObject: fileContent forKey: @"fileContent"];

  [dictionaryService close];

  // store ftconfig.xml in node_ftconfig
  dictionaryService = (id <FTDictionaryService>)
    [node_ftconfig serviceWithId: @"FTDictionaryService"];

  fileContent = [[[NSData alloc] initWithContentsOfFile:
@"ftconfig.xml"]
    autorelease];
  assert( nil != fileContent );
  [dictionaryService setObject: fileContent forKey: @"fileContent"];

  [dictionaryService close];

  [transaction commit];

  NSLog( @"Graph created!" );
  [pool release];
}
```

After integration of this function, your main() function may look like:

```
int main( int argc, char *argv[] ) {
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
  FTBootstrap *bootstrap;
  id <FTAdministrationSession> session;

  NSLog( @"**FTSample " );

  startServer(YES);
  session = (id <FTAdministrationSession>) login( @"altum",
@"silentium" );

  createGraph( session );

  [pool release];

  return 0;
}
```

Summary

Within this sections the creation of graphs has been described. With respect to the current state of FT, you should have seen all functions which are available regarding the creation process at present.

While thinking of this example and potentially about ongoing steps please *keep in mind, that the current state is not a release*, *it is the first snapshot of the implementation*!

Regarding the example: The next step would be to read the graph and to perform an adaequate checkout operation. This is subject of the next section.

8. Reading Graphs

After the creation of a graph, the read operations work quite analogeously. With the API reference provided with the snapshot the reader should be able to find out the corresponding classes, protcols, methods etc. and get information about their usages. That's why the second part of the example, namely the implementation of a checkout operation using the underlying read operation, is left to the interested user. In the following subsections only the description of the tasks to be performed and a first step into the reading of the graph are given.

Steps to Perform Regarding the Example

In the example we stored all information and data (source file, build file, configuration file) used to checkout the example at a different location. Now the main tasks to run a checkout operation are:

1 Ask the user for the new location for the example project (YOU DO NOT REALLY WANT TO USE THE ORIGINAL DIRECTORY!)

2 Iterate through the previously created graph and depending on the node

  2.1 Write the content of the node, which has been stored using the attached dictionary service, into the new location using the appropriate file name OR

  2.2 Create the corresponding directory.

By doing so you will learn:

- How to read nodes and how to "follow" outgoing edges.
- How to read data from a dictionary service.

To give an entry into this task a short piece of code is presented in the next section.

Reading the Example Graph

Before any explanation is given, the first lines of the code useful to checkout `FTSample are presented`:

```
void checkout( id <FTSession> session, NSString *newLocation ) {
  id <FTGraphManager> graphManager;
  id <FTObjectToIdMapper> sysOM;
  id <FTObjectToIdMapper> graphOM;
  id <FTId> graphId;
  id <FTGraph> graph;
  id <FTId> id_application;
  id <FTNode> node_application;

  NSLog( @"Now reading graph content and performing a checkout..." );
  sysOM = [session defaultObjectToIdMapper];
  graphManager = [session graphManager];

  // start transaction

  // Step I: create identifier for graph
  graphId = [sysOM mapObject: @"FTSampleGraph"];

  // Step II: load graph
  graph = [graphManager graphWithId: graphId];

  // Step III: get object mapper of the created graph
  graphOM = [graph objectToIdMapper];

  // Step IV: create the five identifiers:
  id_application = [graphOM mapObject: @"Application"];

  // Step V: get  the application node
  node_application = [graph nodeWithId: id_application];

  // ... TODO
}
```

As you may see, the code is quite straight forward having the creation operation in mind:

I. Create an identifier `ig` for the graph we have to load.

II. Load the graph `g` using the identifier `ig`.

III. Obtain the object mapper `om` of the created graph.

IV. Create the identifier used to identify the node “`Application`”.

V. Fetch the application node

9. Certain Notes about FT

Managing Accounts

At present FT does not support the creation of additional accounts. As a result you only have one account – the admininistration account. This account has the login-id “**altum**” and the password “**silentium**”.

Supported Services

Dictionary Service