Fremont XML Library

The best XML code is code that you don't have to write

Product Documentation

Version: 1.2.0.0

Table of Contents

1. Introduction to Fremont

Probably the best way to introduce Fremont is to answer the standard questions every developer will have.

What is Fremont?

Fremont is a static library that provides the ability serialize Objective C objects to XML documents, and deserialize XML documents to Objective C objects.

Do I have to write any XML parsing code to use Fremont?

No. Fremont handles all XML parsing for you.

How does Fremont understand the relationship between my XML document and my class structure?

You declaratively define this mapping in a *model definition* XML file. Fremont then reads this XML file, and based on your model definition, knows how to create the desired XML document structure during serialization, and how to create the desired Objective C object hierarchy during deserialization.

What iOS versions does Fremont support?

Fremont has been designed to work with iOS 4.

Does Fremont support the mapping the entire W3C XML Specification to Objective C classes?

No, intentionally so. What is legal in XML is not always analog to class structure. Fremont encourages XML structures that will cleanly and logically map to a class hierarchy.

Is there a third-party XML library, such as libxml2, underneath Fremont?

No. Fremont is built entirely on NSXMLParser.

Does Fremont build a DOM or any memory intensive data structure under-the-hood?

No. The only data structures that Fremont creates is an in-memory representation of your model definition, and if deserializing, your target object you are deserializing to. NSXMLParser is essentially a SAX parser, and aside from managing a linear list of parsed XML nodes, which is a standard parsing exercise, there are no other data structures created.

Has Fremont been run through Instruments?

Yes. Fremont has been profiled in Instruments and is memory-leak free.

Is using Fremont difficult?

No. There are essentially three steps:

- Create your model definition XML file.
- Tell Fremont where to find your model definition XML file.
- Serialize or deserialize.

Why would I want to use Fremont?
Fremont eliminates the need to write XML parsing code, which is a significant time-saver in development. One ancillary benefit, is that as it is designed to provide a means to map XML documents to hierarchical object structures, it discourages XML document design that don't gracefully map to objects.

When would I not want to use Fremont?
If you have a legacy XML document structure which needs to be deserialized from / serialized to, the XML document structure contains some custom mapping which Fremont does not support, and/or there are custom operations beyond merely mapping to an object, and there is not the ability to alter the XML document structure, then perhaps it will require doing custom XML parsing / serialization.

What's with the name "Fremont"?
Fremont is named after a mountain peak in Arizona.

What does a Fremont license cost?
Nothing. Fremont, and its source code, are free, and offered under an MIT-style (with minor changes) license. Use of Fremont in an iOS app merely requires visible attribution to Big Hill Software somewhere in the app.

NOTE: For the remainder of this document, please reference the Fremont_Sample sample project, which demonstrates proper usage of Fremont.

2. XML Types

Fremont specifies two XML types:

- ATTRIBUTE: an XML attribute
- ELEMENT: an XML element

Every value is represented as either an XML attribute or element.

3. Objective C Data Types

Fremont supports several fundamental Objective C data types, which are listed with their mappings below:

Type	Description	Objective C Mapping
STRING	a string value	NSString
NUMBER	a numeric value (use for all integer and float numeric values)	NSNumber
BOOLEAN	a boolean value (supports string conversions to boolean values consistent with the NSString boolValue method)	BOOL
DATE	a date value, in a user-defined format, according to format-specifiers for NSDate objects	NSDate
ARRAY	an array that must contain children of a specific data type. An array may not contain children of heterogeneous types. Associated with the ModelArray element in a model definition.	NSArray
DICT	a dictionary that can contain children of heterogeneous data types. Associated with the ModelDictionary element in a model definition.	NSDictionary
DATA	binary data, which within the XML document should be a CDATA type, and should have Base 64 encoding.	NSData
ENUM	a custom enum type. Associated with the ModelEnum element in a model definition.	A custom enum

Type	Description	Objective C Mapping
UDC	a User-Defined Class, a custom class. Associated with the ModelClass element in a model definition.	A custom class

4. Model Definition

A model definition defines the relationship between the structure of an XML document and a class hierarchy. When Fremont serializes an object to XML, it references the model definition to determine how to generate XML from the object and its properties. When Fremont deserializes an object from XML, it references the model definition to determine what objects to instantiate and what properties to set with values read from XML.

Developers express model definitions declaratively in XML. The model definition schema (ModelDefinition.xsd) can be found in the Fremont project (and also can be found in the Fremont_Sample project, but this file is actually a reference to the file in the Fremont project). This schema defines the legal structure of a model definition.

Model definitions have several primary structural elements:

- Model
- Enumeration
- Enumeration item
- Class
- Property
- Array
- Array element
- Dictionary
- DIctionary element

These elements will be discussed in the following sections.

Model

A model (element name: Model) is essentially top of the hierarchy; in the XML document, it is the root element (also called the document element); in the class hierarchy, it is the parent class. The Model element has the following attributes:

- rootDataType: the data type of the root element
- rootXMLName: the XML element tag
- rootFormatXMLString: the format string used to convert date values (this is ignored if not a DATE data type)

Model elements can directly contain ModelEnum, ModelClass, ModelArray, and ModelDictionary elements.

Enumeration

An enumeration (element name: ModelEnum) is a custom defined enum type. ENUM data types are references to ModelEnums. The purpose of using an enumeration rather than a STRING type is to limit the values to a specific set of pre-defined values. The ModelEnum element has the following attributes:

- enumName: the name of the Objective C enumeration
- xmlName: the XML element tag

ModelEnum elements can contain ModelEnumItem elements, which define the values for the enumeration.

Enumeration Item

An enumeration item (element name: ModelEnumItem) defines the values in a custom enum type. The ModelEnumItem element has the following attributes:

- name: the name of the Objective C enum value constant
- value: the integer value associated with the Objective C enum name

Class

A model class (element name: ModelClass) is a custom defined class. UDC data types are references to ModelClasses. The ModelClass element has the following attributes:

- className: the Objective C class name
- xmlName: the XML element tag

- textDataType: the data type of the property associated with element text
- textPropertyName: the name of the Objective C class property associated with element text
- textFormatXMLString: the format string used to convert date values (this is ignored if not a DATE data type) associated with element text

ModelClass elements can contain ModelProperty elements, which define the child properties on the class.

Property

A model property (element name: ModelProperty) defines properties on custom user-defined classes. The ModelProperty element has the following attributes:

- propertyName: the name of the Objective C class property
- dataType: the data type
- xmlType: the XML type, which designates whether the value is represented as an attribute or element in the XML document
- xmlName: the XML element tag
- formatXMLString: the format string used to convert date values (this is ignored if not a DATE data type)

Array

A model array (element name: ModelArray) is an array of child elements. Elements are all of a single data type, that is, child elements cannot be of heterogeneous types. ARRAY data types are references to ModelArrays. The ModelArray element has the following attributes:

- xmlName: the XML element tag

ModelArray elements can contain one ModelArrayElement element only, which defines the type of all child elements contained within the array.

Array Element

A model array element (element name: ModelArrayElement) defines the type of children which may be contained in an array. The ModelArrayElement element has the following attributes:

- dataType: the data type
- xmlName: the XML element tag

- formatXMLString: the format string used to convert date values (this is ignored if not a DATE data type)

Contents of arrays are always represented as elements in XML files. They can never be represented as attributes.

Dictionary

A model dictionary (element name: ModelDictionary) is a dictionary of key-value mappings. Children may be of heterogenous data type. DICT data types are references to ModelDictionaries. The ModelDictionary element has the following attributes:

- xmlName: the XML element tag

ModelDIctionary elements can contain ModelDIctionaryElement elements, which define the types of all child elements contained within the dictionary.

Dictionary Element

A model dictionary element (element name: ModelDictionaryElement) defines the types of all children which may be contained in a dictionary, and their associated keys. The ModelDictionaryElement element has the following attributes:

- dataType: the data type
- xmlName: the XML element tag
- key: the key to map the value to in the dictionary
- formatXMLString: the format string used to convert date values (this is ignored if not a DATE data type)

Contents of dictionaries are always represented as elements in XML files. They can never be represented as attributes.

5. Model Manager

While Fremont contains many classes, a Model Manager is provided as a convenience class to wrap all relevant operations a user needs to perform. Creation of the model definition instance, serialization, and deserialization, are all performed using the ModelManager class.

6. Design Considerations

In designing application code and XML documents to work with Fremont, note the following considerations:

- XML elements intended to be mapped to array children must also have a parent element which maps directly to the array. XML supports elements that may have multiple occurrences of the same element type, in conjunction with arrays of different types. In Fremont, there are no implicit or derived arrays or containers. Each set of array children must have an explicit array element parent to which they belong.
- Arrays must have children of only a single data type, and the number of children is arbitrary. Dictionaries , on the other hand, support children of heterogeneous data types, and each child must be declared. If the collection which needs to be defined contains only single-type objects (such as a collection of employees), and the number of children can vary, use an array. However, if the collection which needs to be defined contains children with differing data types, use a dictionary.
- The Model element in the model definition XML file must correspond to the document root element in the associated XML document.
- When processing an XML document, class properties which aren't declared will not be serialized. Likewise, XML attributes and elements which belong to an element declared as a model class will be ignored when deserializing. In the case of an ignored element, the element and its entire contents are ignored.
- ModelDefinition instances can be reused for multiple serialization / deserialization operations.
- ModelManager instances can be reused for multiple serialization / deserialization operations. However, ModelManager instances are not thread-safe, so there should be only one invocation on a ModelManager object at a time.

7. Caveats

Fremont comes with the following caveats:

- User-defined model definition XML files are not presently validated against the ModelDefinition.xsd file.
- Fremont does not work at present with user-defined classes which are CoreData managed objects. The reason for this is that managed objects require a different approach to creating object instances than non-managed objects. If using CoreData managed objects, you need to create an intermediary adapter class to use with Fremont, from which you can create your managed objects.
- Fremont presently does not pay any attention to name-spacing within XML documents.

8. Use Cases

There are really only two use cases with Fremont: serialization and deserialization. Both use cases require the creation of a ModelManager instance. Both of these use cases are demonstrated in the Fremont_Sample project.

To create a ModelManager instance:

```
modelManager = [[FMXMLModelManager alloc]
initWithModelDefinitionFile:@"ModelDefinition"];
```

To serialize an object:

```
     Company *company;
     ...

     NSError *error = nil;
     self.xml = [modelManager objectToXML:company
withEncoding:NSUTF8StringEncoding error:&error];
```

To deserialize an object:

```
     NSError *error = nil;
     Company *company = [modelManager xmlToObject:xml
withEncoding:NSUTF8StringEncoding error:&error];
```

9. The Fremont_Sample Sample Project

A sample project demonstrating usage of Fremont has been provided, called Fremont_Sample. This project is an iPhone app which can be run in the simulator to demonstrate serialization and deserialization of all data types which Fremont supports. The fundamental data structure used is that of a company, with various properties, multiple computers, and multiple employees with properties. This app performs the following operations:

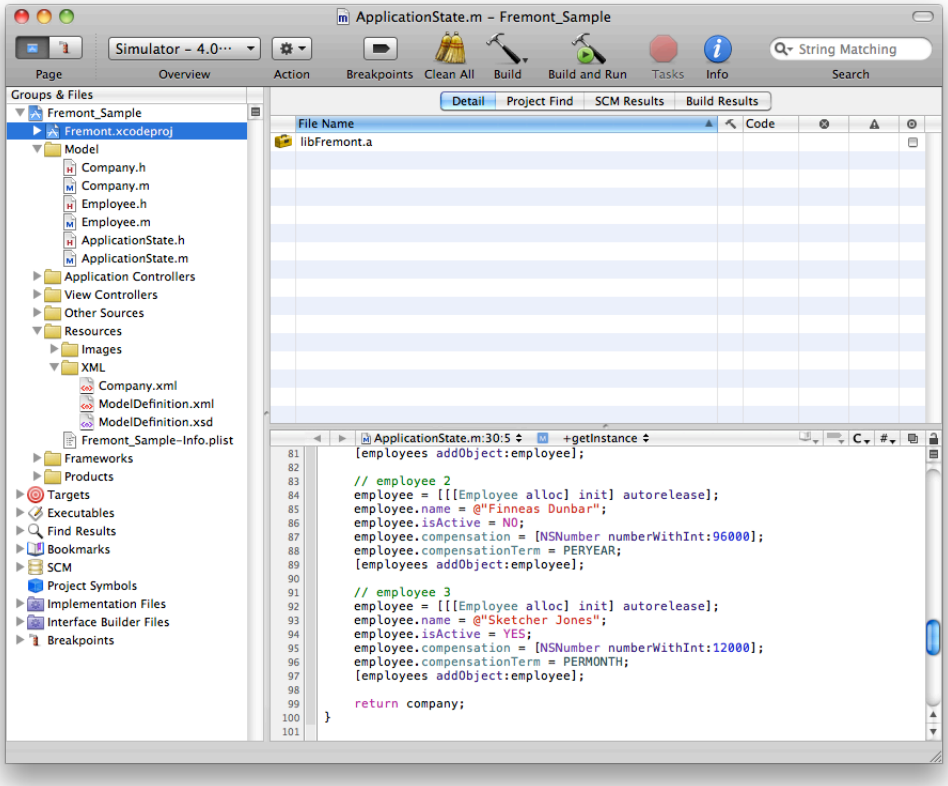
- Create a ModelManager object.
- Create and populate a Company object (including its child objects).
- Serialize the Company object to XML, and set the XML as a state variable.
- Deserialize the XML to a new Company object, and set this object as a state variable.
- Display the serialized XML (state variable) in a view on the screen.
- Display the data in the deserialized Company object (state variable) in a view on the screen.

The ApplicationState class is a good place to start in seeing how this works. Note that the Company.xml file is not actually used by the app, it is just included as a more readable example of the serialized XML which is produced by the app.

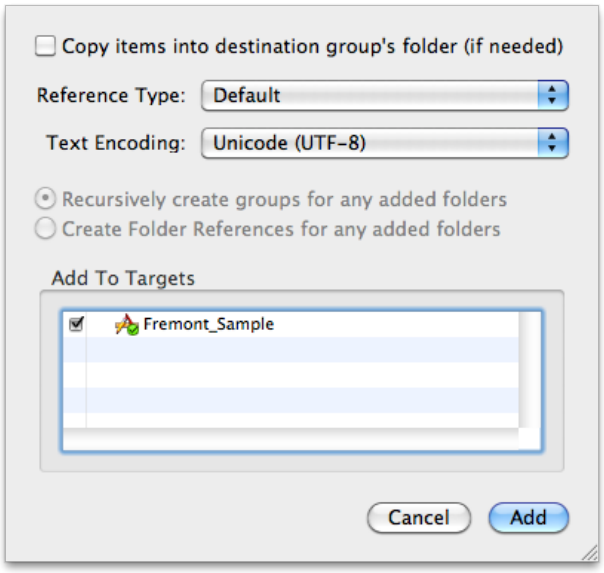
10. Configuring the Fremont Static Library in an iPhone App Project

The following steps will configure an iPhone app project to use Fremont:

1. Add the Fremont Xcode project file to your Xcode project, either by dragging it into your project, or using the context menu on the project in the Xcode Groups & Files pane. Make sure that when the subsequent dialog appears to confirm the action, that the “Copy items into destination group’s folder” checkbox is not checked.

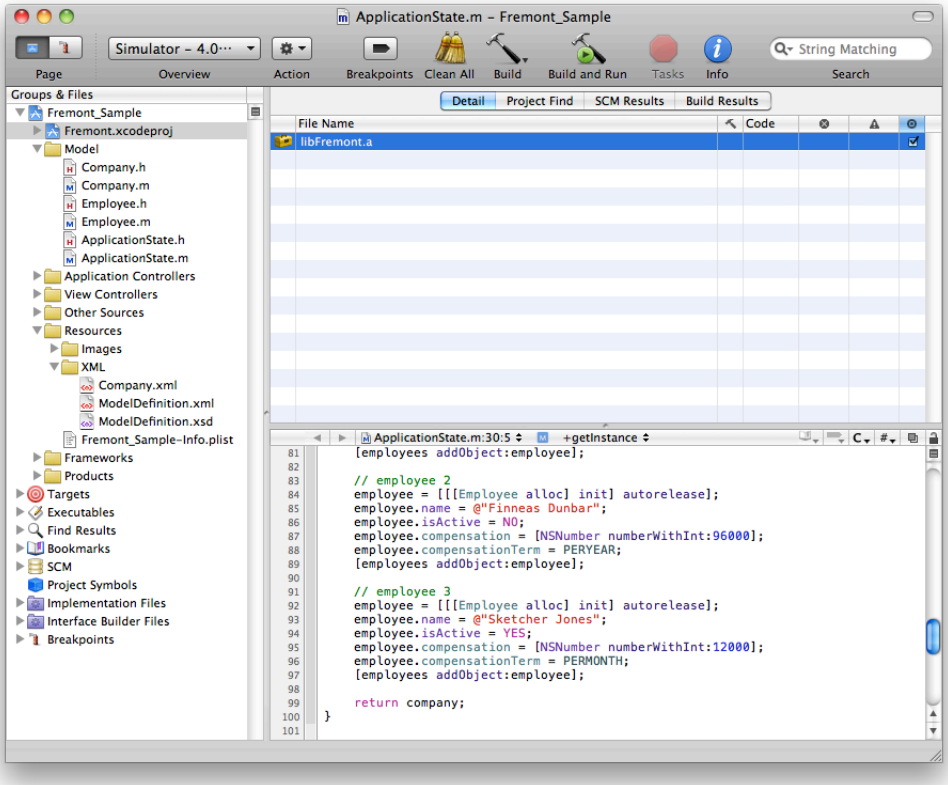


The added Fremont static library is selected



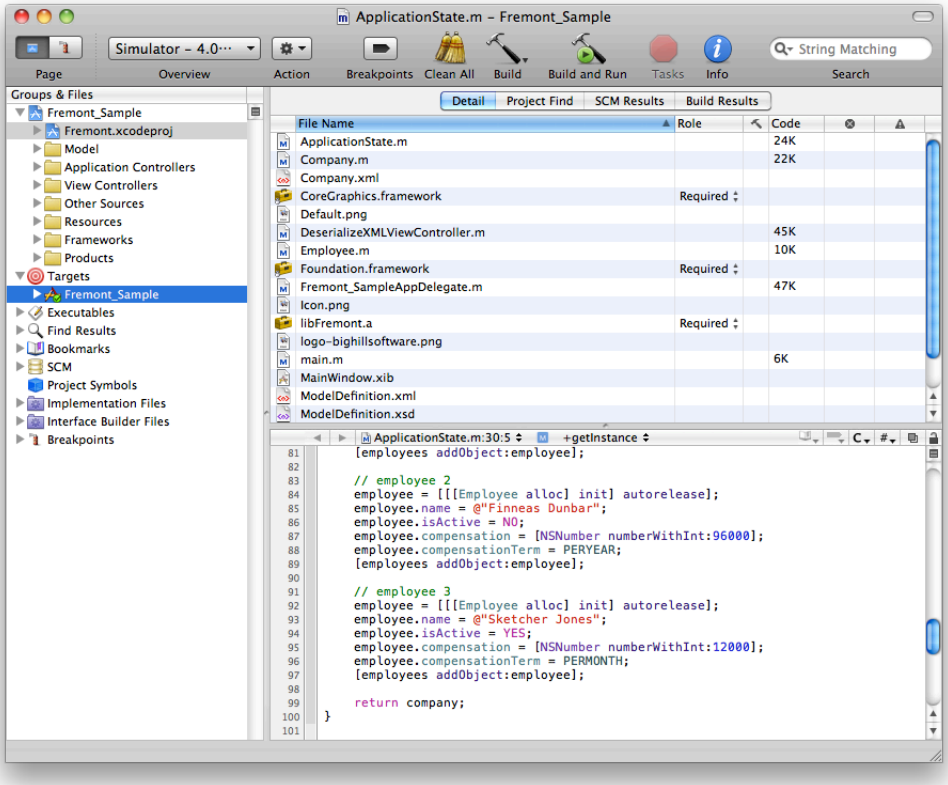
The add dialog, "Copy items..." checkbox not checked

2. Select the checkbox in the Target Membership column of the Detail View.

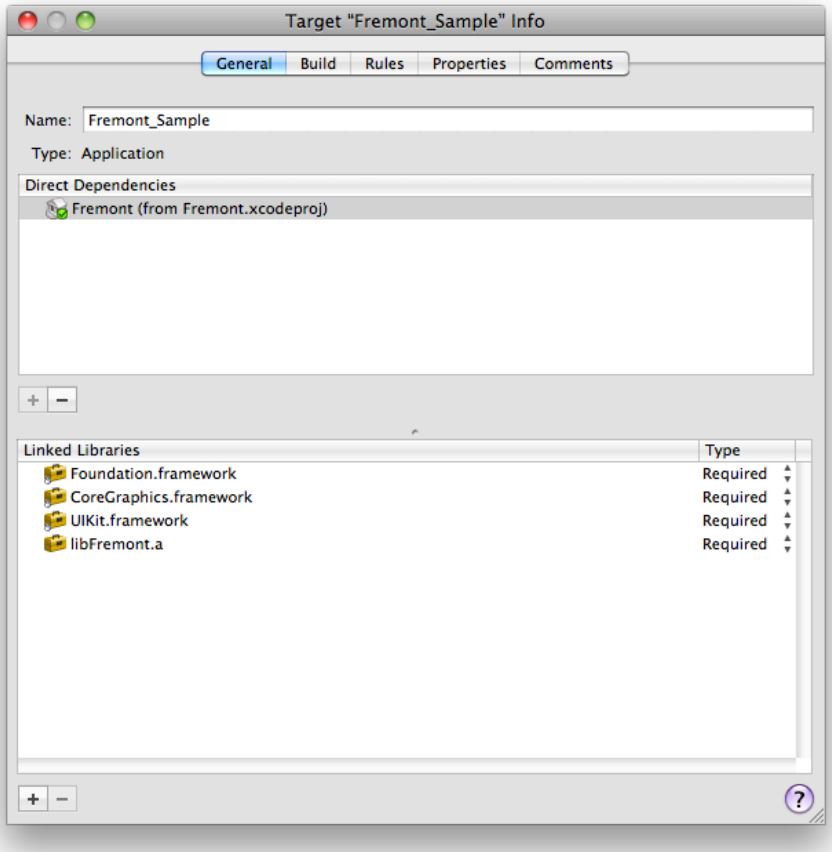


Target Membership checkbox checked

3. Double-click the project target to bring up the target info. Add the Fremont project to the list of Direct Dependencies.



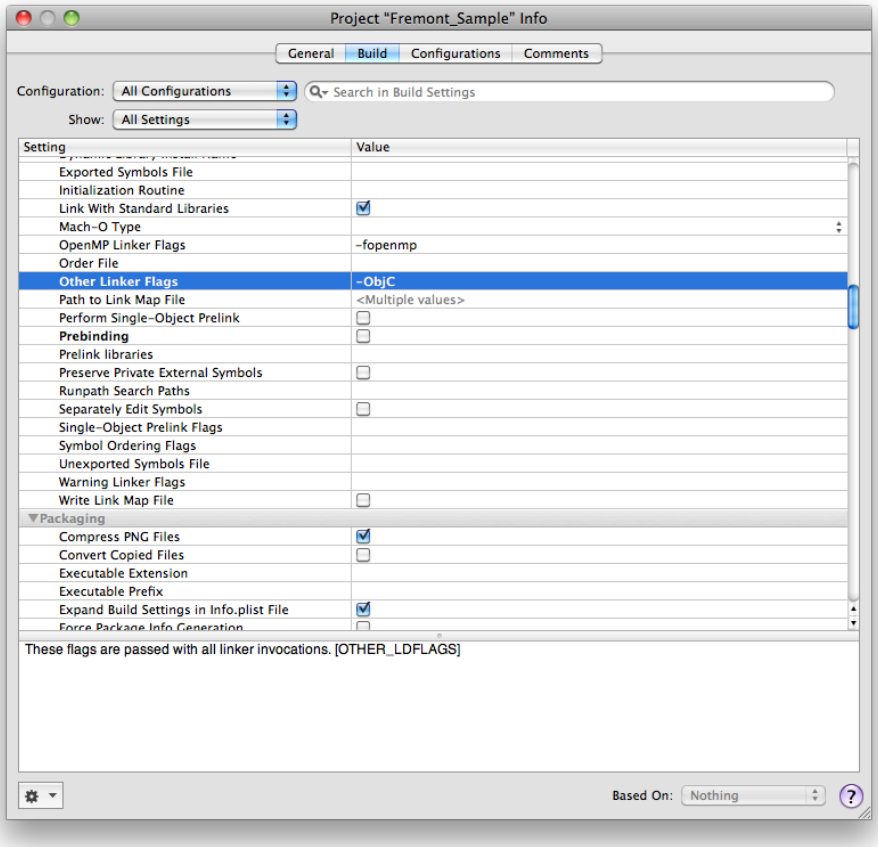
Project target is selected



Fremont added as a Direct Dependency

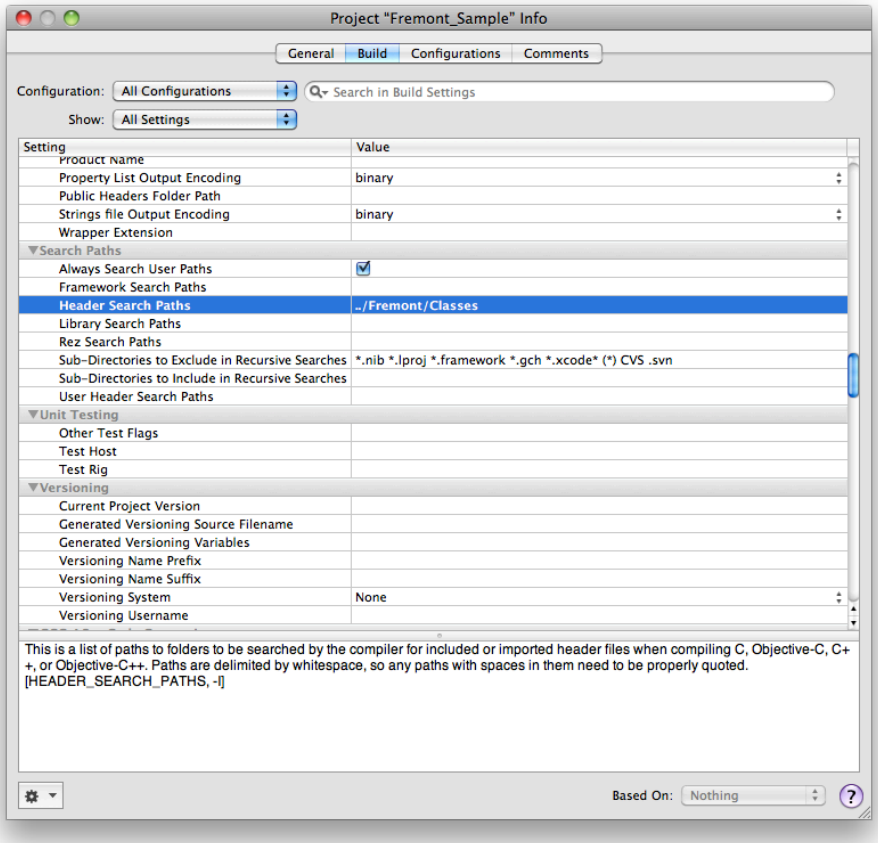
4. Open the project's build configuration, and with the "All Configurations" configuration selected, configure the following two options:

    - Set "Other Linker Flags" to "-ObjC".



"Other Linker Flags" option selected

- Set the “Header Search Paths” option to the appropriate path to the Fremont header files.



“Header Search Paths” option selected

Your project should now be properly configured to use Fremont!