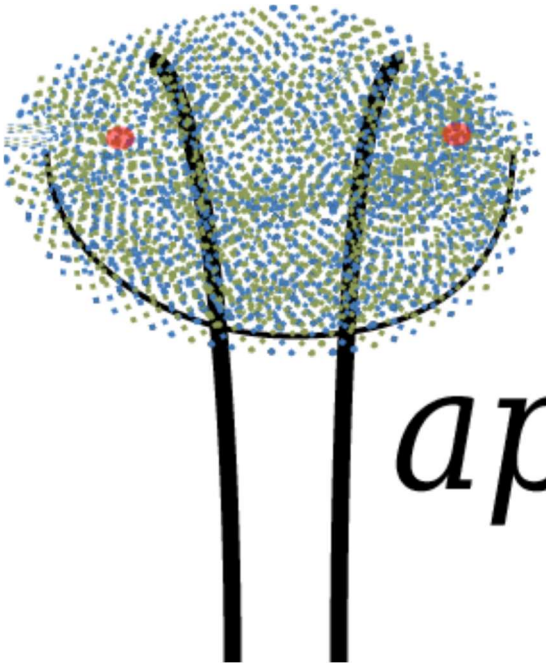


appyTree

Plant your tree of Java objects

Table of Contents

Introduction

Welcome to the official documentation of the HappyTree API v2.0.0.

This document is divided into 3 parts. The first part aims to guide developers through getting started, briefly explaining what the HappyTree API is and its purpose. Additionally, the first code snippets are shown.

Next, this document will present a functional perspective. Here, usage, basic concepts, contexts, and lifecycles are introduced.

Finally, we will show through architectural specifications how the HappyTree API works. This is an in-depth approach that aims to present API details relating to structural composition, behavior, and more technical aspects.

Getting Started

What is it?

HappyTree is a data structure API designed for handling Java objects that have a tree-like behavior, where an @Id attribute of an object is referenced as a @Parent attribute of its children.

In certain circumstances there is a need to convert a list of Java objects, which could represent a model layer in a business context, into an actual hierarchical tree structure in memory, where objects contain their children and each child contains its own children collection and so on.

When there is a need to organize a collection (Set/List) of objects, where each object relates to another object of the same type through an identifier attribute, in a tree-like manner, the HappyTree API is able to transform this structure into an actual tree structure in memory, where each object will be wrapped into a tree node object, called Element, and each element contains its children elements where each child contains its own children and so on, recursively.

From this point, the API client can handle those elements within a tree, such as adding/removing children from nodes, moving nodes to another point of the tree or even to another tree, copying nodes to other trees, converting trees into JSON/XML, etc.

Therefore:

"HappyTree API is a data structure API designed for the Java programming language that consists of transforming linear structures of Java objects into a tree structure and allowing their handling."

What is the objective?

The HappyTree API aims to provide a way of creating new trees, creating trees from an existing collection of objects that have tree-like behavior, as well as for handling these trees. It provides interfaces for the API client for three primary and clear objectives:

- Handle Java objects as if they were nodes within trees to perform operations such as copying, cutting, removing, creating, persisting/updating, etc. on those objects.
- Transform linear data structures of Java objects that have tree-like behavior into an actual tree.
- Create new trees from scratch.

The first purpose represents the basic operations of the trees, when the API client desires to change the state of the nodes (officially called Element in the context of the API) in the trees, to move, copy, remove, create and update those nodes.

The second purpose is suitable for situations in which the API client needs to transform a collection of plain objects, which have a logical tree relationship between them, into an actual tree. Here, each element contains its collection of child elements, and each child from this collection contains its own children recursively.

The last one allows the API client to create new trees from scratch, persisting element to element to build the tree structure as desired.

When to use?

This is useful when the developer feels the need to handle objects that have a tree-like behavior in their applications. There are several scenarios in which this API can be useful, such as:

- Handling directory structures.
- Handling organizational structures.
- Handling visual component structures.
- Handling product category structures.
- Handling comment/reply-to-comments structures.
- And many other scenarios.

For the above scenarios, when the developer already has a previous collection of objects in which those objects are only linearly referenced by each other, this API has precisely this purpose of transforming this linear structure into a physical tree structure. This process is known as the **API Transformation Process**, and it is one of the main core functionalities of the HappyTree API.

If, for example, the project has a collection of Java model objects representing directories in a file system, where each directory object has an identifier attribute which is referenced by the parent attribute from another directory object in the same collection, then this API can be used to transform this linear structure into an actual tree structure in memory.

Suppose we have something like this:

```
//Linear tree structure.
public class Directory {
	//Own ID
	private Integer dirId;
	//Super node reference
	private Integer dirParentId;
	//Simple attribute
	private String dirName;

	//getters and setters
}
```

But we want this:

```
//Recursive tree structure wrapped through the Element object.
public interface Element<Directory> {
	private Object id;
	private Object parentId;
	private Collection<Element<Directory>> children;

	//Skeleton methods.
	public void addChild(Element<Directory> child);
	public void removeChild(Element<Directory> child);
	public void wrap(Directory directory);
	public Directory unwrap();
}
```

Notice that in the first Directory class; to reference one instance to its parent, it is necessary to have an implementation that binds the respective "*dirParentId*" attribute with the "*dirId*" from another instance. Here, the objects are related among themselves in a linear way, with no list of children or operations to deal with them. It is just a simple Java POJO.

The second Element class is now the ideal way to represent a node within a tree, as each instance of the Element class will structurally have its own children collection. Also, it can "store" the real object from the first example, the Directory class, in other words, this object is what is called a "**wrapped object node**". Notice that one instance of the Element now has its collection of children as well as a set of operations that allow the API client to handle them.

Requirement

>= Java 8

Importing

To import the HappyTree API for inside of a Java project, copy one of the following codes:

Maven

```
<dependency>
    <groupId>com.madzera</groupId>
    <artifactId>happytree</artifactId>
    <version>2.0.0</version>
</dependency>
```

Gradle

```
implementation 'com.madzera:happytree:2.0.0'
```

Note

Maven

Compared to **v1.0.0**, the **groupId** of this new version has changed:

```
- <groupId>com.madzera.happytree</groupId>
  <artifactId>happytree</artifactId>
- <version>1.0.0</version>
+ <groupId>com.madzera</groupId>
  <artifactId>happytree</artifactId>
+ <version>2.0.0</version>
```

Gradle

```
- implementation 'com.madzera.happytree:happytree:1.0.0'
+ implementation 'com.madzera:happytree:2.0.0'
```

Usage

To demonstrate how to use it, let's consider a practical exercise common in several projects: a simple menu structure. Suppose we received a ticket to adjust the system menu where a submenu item needs to be relocated to another menu category, and all menu items are stored in a relational database. We should have something like this:

MENU_ID	MENU_LABEL	MENU_PARENT_ID	MENU_DESCRIPTION
105	Administration	null	
110	Control Panel	105	...
302	Users	null	
321	My Profile	302	...
322	Access Control	302	...

The purpose of the ticket would be to relocate the "*Access Control*" menu to stay within the "*Administration*" menu, in other words, it would move from the "*Users*" menu to the "*Administration*" menu.

However, because it is a legacy project, the development team did not take the necessary care, and when loading this structure from the database to the respective Java Menu objects, the development team did not physically treat this entire structure as tree menus. Therefore, the object in question looks like this simple POJO:

```java
public class Menu {
	private Integer menuId;
	private String menuLabel;
	private Integer menuParentId;
	private String menuDescription;

	//Default constructor and getters & setters.
}
```

As each object of the class above represents a menu item, we do not have here, in terms of object-oriented programming, a defined tree structure, but rather a structure that matches the way it is stored in the database, that is, a relational/linear structure.

But this is not what is intended, because in addition to the structure not being physically like a tree, some extra work will probably be necessary to implement recursive methods and other methods to perform operations on the nodes of the menu tree. Therefore, this would be a good situation to use the HappyTree API.

The above structure would be transformed by the HappyTree API (through the **API Transformation Process**) into:

```
public interface Element<Menu> {
        private Object id;
        private Object parentId;
        private Collection<Element<Menu>> children;
        private Menu wrappedNode;

        //Skeleton methods.
        public void addChild(Element<Menu> child);
        public void removeChild(Element<Menu> child);
        public void wrap(Menu menu);
        public Menu unwrap();

        //Other methods.
}
```

With the transformation performed, each Element object encapsulates (wraps) its respective Menu object within itself, and each Element object is physically positioned in the tree, thus representing a tree node.

In addition, each element can have several other elements within it as children, and each child can have other children, and so on, recursively representing a complete tree.

After the tree is built, you can relocate the desired menu item using the interfaces provided by the HappyTree API, without the need to implement any additional code.

To solve the ticket, it is necessary to put some Java annotations and implement Serializable in the Menu class:

```
@Tree
public class Menu implements Serializable {
        @Id
        private Integer menuId;
        private String menuLabel;
        @Parent
        private Integer menuParentId;
        private String menuDescription;

        //Default constructor and getters & setters
}
```

@Tree

It indicates that an object in this class can represent a node within a tree. It is useful when there is a collection of objects that this class annotates to be converted (transformed) automatically into nodes within a tree. This process is known as the **API Transformation Process**.

@Id

Unique and non-null identifier of the object to be transformed.

@Parent

Identifier of the parent object to which the current object will bind during transformation.

There are some conditions for the **API Transformation Process** to be successful:

- The three annotations must be present in the class to be transformed.
- The value of the attribute annotated by the @Id must be mandatory, while the attribute annotated by the @Parent can be null, or point to a non-existent parent. This @Parent attribute is responsible for moving or not moving the object to the root level of the tree, depending on whether it is null or not found.
- The attribute annotated by the @Id must be of the same type as the attribute annotated by the @Parent.

From this point on, after just adding these annotations to the class attributes, we have everything we need to transform this linear structure into a real tree structure. We now have the base for our first code snippet.

First code snippet

To initialize the menu tree in the example above, and any other type of tree, we use a code snippet that is quite common and will always be used at any tree initialization:

```
public void foo() throws TreeException {
       Collection<Menu> menus = myObject.getMenuFromDatabase();
       TreeManager manager = HappyTree.createTreeManager();
       TreeTransaction transaction = manager.getTransaction();
       transaction.initializeSession("MyFirstHappyTree", menus);
}
```

From the code above, the tree is already built and has a session identifier named "*myFirstHappyTree*". Every initialized tree (session) has a unique and non-null session identifier. We will discuss these concepts in more detail later.

However, it remains to fulfill the objective of the ticket. Although the tree is already built, it is still necessary to reallocate the "*Access Control*" menu from "*Users*" to "*Administration*".

As we already know, through the database in the example above, the menu item with the label "*Access Control*" has the @Id 322 and the menu item "*Administration*" has the @Id 105. With that in mind, below is the code to relocate the menu item:

```
public void foo() throws TreeException {
	Collection<Menu> menus = myObject.getMenuFromDatabase();
	TreeManager manager = HappyTree.createTreeManager();
	TreeTransaction transaction = manager.getTransaction();
	transaction.initializeSession("MyFirstHappyTree", menus);

	Element<Menu> administration = manager.getElementById(105);
	Element<Menu> accessControl = manager.getElementById(322);
	manager.cut(accessControl, administration);

	//Alternatively, this also can be used
	//manager.cut(322, 105);
}
```

Now, the ticket has been solved. However, this practical example used a collection of objects that represent menu items from a database. Another possibility is to build the same tree from scratch, creating it element by element until the resulting tree is ready

```
public void foo() throws TreeException {
	final Integer admId = 105;
	final Integer accessId = 322;

	Menu adm = new Menu();
	Menu ac = new Menu();

	adm.setMenuId(admId);
	adm.setMenuLabel("Administration");
	ac.setMenuId(accessId);
	ac.setMenuParentId(admId);
	ac.setMenuLabel("Access Control");

	TreeManager manager = HappyTree.createTreeManager();
	TreeTransaction transaction = manager.getTransaction();
	transaction.initializeSession("MyFirstHappyTree", Menu.class);

	Element<Menu> administration = manager.createElement(admId, null, adm);
	Element<Menu> accessControl = manager.createElement(accessId, admId, ac);

	administration.addChild(accessControl);

	//In fact, this saves the new element within the tree
	administration = manager.persistElement(administration);
}
```

The most important difference compared to the first example is the line *“transaction.initializeSession("MyFirstHappyTree", Menu.class);”*. Here, the class type is specified instead of a collection of Menu objects, because the tree is being built from scratch. In this case, the API client needs to create elements one by one and, in the end, use the *“manager.persistElement(administration)”* method to save the new elements.

Functional Overview

Basic Concepts

HappyTree API is a Java library that helps Java developers handle data structures with hierarchical behavior. It is a simple and small library; despite this, it is still considered an API because it has a set of rules and validations that are applied to the interface's methods provided to the API client. The HappyTree API also implements lifecycle concepts regarding the elements that represent tree nodes, as well as in the **API Transformation Process**.

Furthermore, contexts are also applied within the HappyTree API, because when the API client obtains an element from a tree, it is obtaining a copy of that element so that it can edit it and then update the tree.

As can be observed, there is a whole set of rules, validations, and lifecycle concepts, as well as contexts, that define HappyTree as an API, even though it is a small Java library.

With this in mind, understanding these concepts becomes necessary before we move on to the architectural specification of the HappyTree API. Here, these concepts will be presented in a basic way, but they will be explored in greater depth throughout the next chapter.

Interfaces & Classes

Interface	Description
Element	Represents a node within a tree.
TreeManager	Responsible for providing operations to the API client.
TreeSession	It is the tree. It stores all elements within the tree.
TreeTransaction	Stores sessions but it is only capable of working on one session at a time.
HappyTree	Entry point class.
TreeException	Exception class of the HappyTree API.

Element

An element represents a node in a tree. It can have none or many other elements within the tree, such as children, and each child, likewise, can have several other elements, and so on.

In addition, each element has a unique and non-null @Id and a nullable @Parent, representing the parent identifier which the element references. If the parent is not found or is null, then the element will stay at the root level of the tree.

This object has a defined lifecycle, which will be explained later.

TreeManager

Object responsible for performing operations on trees. It is through the TreeManager that it is possible to create, cut, copy, remove, update and persist elements over a given tree session that was selected through a transaction.

All TreeManager operations need a transaction referencing an active session, otherwise a TreeException will be thrown.

Therefore, all these interfaces are related in the following way:

TreeManager (invokes) -> TreeTransaction (to store) -> TreeSession (that contains) -> Element

TreeSession

A session is nothing more than a tree, containing all the elements. The session is represented by the TreeSession interface and must contain a unique (and not null) String identifier, considering that the API client may contain multiple sessions. A session can have 3 states:

State	Description
Activated	The tree exists and can be handled.
Deactivated	The tree exists and cannot be handled.
Destroyed	The tree does not exist anymore.

TreeTransaction

A transaction is an object that is always linked to the TreeManager interface instance and is responsible for managing the various sessions (trees) that the API client may have. It is represented by the TreeTransaction interface, and its main function is to perform management operations on sessions (initialize, deactivate, activate, and destroy).

Although it manages all API client sessions, a transaction can only work on a single session at a time; that is, the transaction can function as a session selector by invoking the "`transaction.sessionCheckout(sessionIdentifier)`" method. The session to which the transaction is currently pointing is known as the **current session**. The methods of the TreeTransaction interface that perform operations on sessions and that do not have parameters (session identifier parameter) are applied directly to the current session. If the **current session** has not yet been selected, null is returned. It occurs in the following methods:

- destroySession().
- activateSession().
- deactivateSession().
- currentSession().

To perform any operation on the TreeManager interface, it is mandatory that the transaction associated with the manager has a session and that this session is active; otherwise, a TreeException will be thrown.

HappyTree

Final class and not instantiable. This class is the one that provides the initial starting point to use the HappyTree API. It is only intended to return instances of TreeManager.

TreeException

Exception class that is thrown by HappyTree API. This exception can be thrown by the TreeManager and TreeTransaction interfaces if any rule or validation is not met.

API Transformation Process

As mentioned earlier, there are two ways to create new trees: from an existing collection of linear objects that represent the tree nodes (whose classes have the @Tree**,** @Id, and @Parent annotations); or from a tree built from scratch, where the tree is constructed manually.

The **API Transformation Process** occurs in the first situation. This is precisely the act of transforming a structure (collection) of linear objects, where the objects are related to each other through identifiers (@Id and @Parent), into an actual tree in memory.

The **API Transformation Process** is automatically triggered when "`transaction.initializeSession("My Tree Session ID", myObjects)`" is invoked, where "*myObjects*" is the collection of objects to be transformed into nodes within a tree.

Root Element

When initializing a session (tree), whether through the **API's Transformation Process** method or manually creating a tree from scratch, it is the sole and exclusive responsibility of the HappyTree API to create the root element.

The root element, as the name suggests, is the first element in the tree hierarchy, corresponding to the parent of all other elements. Being a special element that can only be created by the HappyTree API itself, the root element is simply an element like all the others, with its children below it.

The conceptual difference between a root element and the elements below it is that a root element obviously does not have @Id, @Parent, or the **wrapped object node**. Therefore, some operations involving the Element interface cannot be applied to the root element, such as:

- setId().
- setParent().
- wrap().

Since the root element does not have these three properties mentioned above, the respective *getters* always return null.

From the perspective of the TreeManager interface, some operations are not allowed for root elements, throwing a TreeException. These operations are:

- copy().
- cut().
- removeElement().
- persistElement().

The reason for this is that it doesn't make sense to use the "`manager.cut(source, target)`", "`manager.removeElement(element)`" or "`manager.copy(source, target)`" operations for the root elements. To copy the data from the root element, the API client can invoke "`transaction.cloneSession(from, to)`" as it has the same purpose, that is, copying the entire tree. For the "`manager.persistElement(element)`" method, it also doesn't make sense to use it for the root element because this method should only be used for new elements to be persisted within an already existing tree.

Contexts

This consists of a fundamental concept that determines the perspective from which the API client is currently using the Element objects. These perspectives stem from the fact that when the API client retrieves an element from the tree, it is obtaining a copy of that element. This is as if the original tree was "mirrored" exclusively for the API client, so that elements of this tree can be modified by the API client and subsequently synchronized with the original tree. This synchronization is nothing more than the act of updating the original tree according to the changes made by the API client.

There are two contexts: the **Session Context** and the **API Client Context**.

Session Context

The **Session Context** represents the context that stores the actual trees of the API client. It's where the tree is actually stored, and to modify this tree, the only available way is through the TreeManager interface, provided by the HappyTree API to the API client. When using any method of this interface that makes a direct change to the tree, this change will be applied immediately, without the need to explicitly update the tree by invoking the "`manager.updateElement(element)`" method.

Imagine there's a "box" of trees, where each tree is a session. Therefore, the **Session Context** corresponds precisely to this "box," and everything outside this box is outside the **Session Context**.

API Client Context

This refers to everything outside the **Session Context**. In other words, when the API client retrieves an element, any changes to that element will not be applied immediately. It's necessary to synchronize the change in the original tree for it to take effect. To do this,

simply invoke "`manager.updateElement(element)`" to synchronize the element to apply the change, or "`manager.persistElement(element)`" to add a new element to the tree.

Lifecycle

The HappyTree API implements lifecycle concepts in two places: in Element objects and in the **API Transformation Process**. The explanation of the **API Transformation Process** lifecycle will not be discussed here, only in the next chapter, because the implementation of this lifecycle is not relevant to this section, considering that it is a transparent process for the API client, and because it is an architectural concept of the API itself.

The concept of element lifecycle is closely linked to context, as it is the context that determines whether an element is currently within the original tree or whether it is susceptible to being changed by the API client. Depending on the element's lifecycle state and which method is invoked in the TreeManager interface, a TreeException may be thrown. This occurs because some operations only allow specific element states.

There are 3 states in the lifecycle of an element:

Element Lifecycle

NOT_EXISTED

This is the state when the element is new to the tree session. For this element to be included in the tree, it is necessary to invoke the "`manager.persistElement(element)`" method and thus pass it to the **ATTACHED** state. To create a new element in a tree session, simply invoke the method "`manager.createElement(objId, objParentId, obj)`".

ATTACHED

This indicates that the element is synchronized with the original tree. An element is in this state when the **API Transformation Process** is executed, and after invoking the methods "`manager.persistElement(element)`" and "`manager.updateElement(element)`".

DETACHED

This occurs when the API client modifies an Element object that was previously obtained. The API client can modify the element in such a way as to:

- Change its @Id.
- Change its @Parent.
- Unwrap the object node from the Element object and wrap it again.
- Add and remove children.

Note: the list above does not apply to root elements.

Persist Element

The "*manager.persistElement(element)*" method should only be called when the element is new (**NOT_EXISTED** state) to the current session's tree. To persist, the element as well as all its descendants must also be new. Otherwise, a TreeException will be thrown.

Update Element

When the API client obtains an element from the tree through the TreeManager interface, such as by invoking the "*manager.getElementById(id)*" method, and then modifies it or one of its descendants, to make the change effective it is necessary to "commit" the changes using the "*manager.updateElement(element)*" method.

If the element or one of its descendants has a state other than **ATTACHED** or **DETACHED** (precisely **NOT_EXISTED**), a TreeException will be thrown.

There is a possibility that the API client might retrieve elements but not modify them (**ATTACHED** state). In this scenario, the "*manager.updateElement(element)*" method doesn't throw a TreeException but also doesn't do anything.

Interface Methods

Below are the lists of each interface and its respective methods and descriptions.

Element

Interface	Description
`getId()`	Obtains the element identifier. This identifier is unique within the tree session when attached to the tree.
`setId(Object id)`	Sets the element identifier. The change requires updating the element to take effect. The @Id must be unique and non-null.
`getParent()`	Obtains the parent identifier of this element.
`setParent(Object parent)`	Sets the parent identifier reference of this element. If null or nonexistent, the element will be at root level when persisted/updated.
`getChildren()`	Obtains all child elements of the current element. This includes all descendants recursively.
`addChildren(Element<T> child)`	Adds a new child element into the current element. If the child contains children, they will also be added.

`addChildren(Collection<Element<T>> children)`	Adds a list of child elements to be concatenated to the current children list. Includes all nested children recursively.
`getElementByid(Object id)`	Searches within the current element for an element according to the @Id parameter. Returns null if not found. Search is performed recursively.
`removeChildren(Collection<Element<T>> children)`	Removes a subset of elements within this one. All children and elements below the hierarchy are also removed recursively.
`removeChild(Element<T> child)`	Removes the specified child element from the children list. All its children and elements below are also removed recursively.
`removeChild(Object id)`	Removes the element from the children list by @Id. The element and all its children are removed.
`wrap(T object)`	Encapsulates any object node within the element, as long as it has the same class type as other objects in the same tree session.
`unwrap()`	Returns a copy of the object node wrapped in this element. Provides access to the encapsulated object.
`attachedTo()`	Returns the TreeSession instance to which this element belongs. An element is always associated with a session.
`lifecycle()`	Returns the current lifecycle state of this element (**NOT_EXISTED**, **ATTACHED**, or **DETACHED**).
`toJSON()`	Converts the whole element structure into a JSON format (minified). This includes all children recursively.
`toPrettyJSON()`	Converts the whole element structure into a well-formatted JSON string. This includes all children recursively.
`toXML()`	Converts the whole element structure into an XML string (minified). This includes all children recursively.
`toPrettyXML()`	Converts the whole element structure into a well-formatted XML string. This includes all children recursively.
`search(Predicate<Element<T>> condition)`	Searches for elements that satisfy a specific condition within this element and its children recursively. Returns a list of matching elements with their hierarchical structure preserved.

`apply(Consumer<Element<T>> action)`	Applies a function to be performed on this element and all its children recursively. Changes are not automatically reflected and require persist/update operations.
`apply(Consumer<Element<T>> action, Predicate<Element<T>> condition)`	Applies a function to be performed on elements that satisfy the specified condition within this element's subtree. Changes require persist/update operations.

TreeManager

Interface	Description
`cut(Element<T> from, Element<T> to)`	Cuts the source element to inside of the target element, whether for the same session or not (they must have the same class type). All children of the source element will be cut as well. If target is null, moves to root level.
`cut(Object from, Object to)`	Cuts an element identified by its @Id and moves it inside another element within the same session. Returns null if the source element is not found. If the target element is null or not found, the source element is moved to the root level.
`copy(Element<T> from, Element<T> to)`	Copies the source element into the target element in another tree session. The entire structure of the copied element will be pasted inside the target element. Copying within the same tree is not allowed, as it would result in a duplicate @Id exception.
`removeElement(Element<T> element)`	Removes the corresponding element from the tree session and returns the removed element. After removal, the element and all its children will have the **NOT_EXISTED** state.
`removeElement(Object id)`	Removes the element by its @Id. All children of the found element are removed as well and return the removed element itself. Returns null if @Id cannot be found.
`getElementById(Object id)`	Returns the element given its @Id in the tree session. Returns null if the @Id is null or the element cannot be found in the tree.
`containsElement(Element<T> parent, Element<T> descendant)`	Verifies whether the parent element contains inside of it the descendant element in the current session. Returns false if elements are null or not attached.
`containsElement(Object parent, Object descendant)`	Verifies whether the parent element (identified by @Id) contains the descendant

	element (identified by @Id) within the current session. Returns false if either element is not found.
`containsElement(Element<?> element)`	Verifies that the current tree session has the specified element. Returns false if element is null, not found, or not in **ATTACHED** state.
`containsElement(Object id)`	Verifies that the current tree session has the specified element by the given @Id. Returns false if element is not found or @Id is null.
`createElement(Object id, Object parent, T wrappedNode)`	Creates an element with the @Id, parent and the **wrapped object node**. Returns a new element with the **NOT_EXISTED** state in lifecycle. Must be persisted to be added to the tree.
`persistElement(Element<T> newElement)`	Persists a new element into the current tree session. The new element must have a unique identifier and **NOT_EXISTED** state. Returns a copy with **ATTACHED** state.
`updateElement(Element<T> element)`	Updates the state of the element within the tree. Returns a copy with **ATTACHED** state.
`getTransaction()`	Obtains the TreeTransaction instance associated with this manager. Every operation defined in this interface needs to check the transaction.
`root()`	Returns the root of the tree in the current session. The root encompasses all other elements and has no @Id, @Parent, or **object wrapped node**.
`search(Predicate<Element<T>> condition)`	Searches for elements that satisfy a specific condition within the entire tree structure. Returns a list of elements (including their children) that match the condition.
`apply(Consumer<Element<T>> action)`	Applies a function to be performed on all elements within the entire tree structure (except root). Changes are automatically reflected on the tree session.
`apply(Consumer<Element<T>> action, Predicate<Element<T>> condition)`	Applies a function to be performed on elements that satisfy a specific condition within the entire tree structure (except root). Changes are automatically reflected on the tree session.

TreeSession

Interface	Description
`getSessionId()`	Returns the session identifier name. A session identifier is defined when the session is initialized and must be unique.
`isActive()`	Verifies whether the session is active. Returns true if the session is active (can be handled), false if deactivated (exists in memory but cannot be handled).
`tree()`	Returns the entire tree session structure, represented by the root element. From the root element, it is possible to navigate through all children recursively, accessing the entire tree structure.

TreeTransaction

Interface	Description
`initializeSession(String identifier, Class<T> type)`	Initializes a new empty tree session with the specified identifier. Creates an empty tree where the API client must create elements one by one. The session is automatically checked out as the current session.
`initializeSession(String identifier, Collection<T> nodes)`	Initializes a session with a specified identifier and transforms a list of linear objects (with logical tree structure) into an actual tree structure through the API Transformation Process. The session is automatically available as the current session.
`destroySession(String identifier)`	Removes the session with the specified identifier permanently. The tree and its elements within this session are also removed and cannot be retrieved.
`destroySession()`	Removes the current session permanently. The tree and its elements within this session are also removed. The API client needs to specify a new session to be checked out after removal.
`destroyAllSessions()`	Removes all registered sessions permanently. The removal occurs for both activated and deactivated sessions.
`sessionCheckout(String identifier)`	Selects a tree session to work with . The current session remains in the background while the checked-out session becomes the current session. Passing null or non-existent identifier cancels the current session.

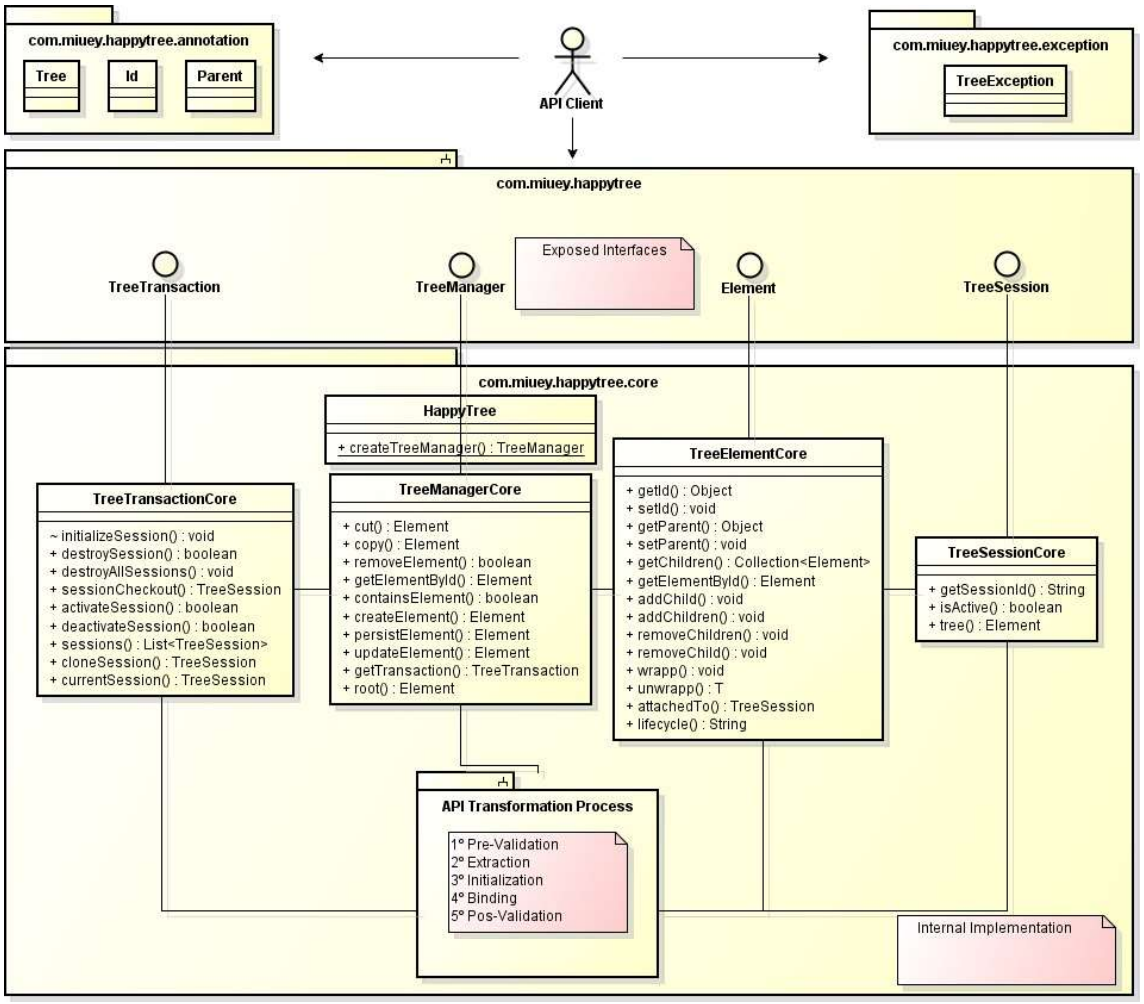
`activateSession(String identifier)`	Activates a session by the specified identifier. With an active session, its elements can be handled freely within the tree. It does not automatically make it as the current session.
`activateSession()`	Activates the current session. With an active session, its elements can be handled freely within the tree. The current session will always be active after invoking this method.
`deactivateSession(String identifier)`	Deactivates a session by the specified identifier. The session is just disabled but not removed. With a deactivated session, its elements cannot be handled.
`deactivateSession()`	Deactivates the current session. The session is just disabled but not removed from the list of registered sessions. With a deactivated session, its elements cannot be handled.
`sessions()`	Returns the list of all registered sessions. The list includes both activated and deactivated sessions.
`cloneSession(String from, String to)`	Replicates the tree session defined by the "*from*" identifier to the session defined by the "*to*" identifier. Faithfully reproduces all elements from source tree to target tree. If target exists, it is replaced. It does not automatically check out the cloned session after the cloning process ends.
`cloneSession(TreeSession from, String to)`	Replicates the tree session defined by the "*from*" session instance to the session defined by the "*to*" identifier. Faithfully reproduces all elements from source tree to target tree. If target exists, it is replaced. It does not automatically check out the cloned session after the cloning process ends.
`currentSession()`	Returns the current session of the transaction. The current session is the one that the transaction is referring to at this moment. Returns null if no session is checked out.

Architecture Overview

Although the HappyTree API can be used with relative ease, reading this chapter is recommended to gain a complete understanding of all aspects of the API.

A top-down approach is adopted, starting with an architectural overview and an explanation of the structural and behavioral composition of the HappyTree API, and concluding with the technical details.

In the image below, the architecture overview of the HappyTree API is presented, along with its class packages and their respective responsibilities.



The HappyTree API consists of several packages, but two of them can be considered main:

- **com.madzera.happytree**

  This is the package through which the API client can view and use the exposed interfaces. It contains only interfaces that must be exposed as functionalities. **Therefore, everything contained in this package must be public and accessible to the API client**.

This package contains the following interfaces:

- Element.
- TreeSession.
- TreeTransaction.
- TreeManager.

- **com.madzera.happytree.core**

  This package contains the actual implementations of the exposed interfaces, in addition to implementing the Element lifecycle (discussed later) and the phases of the **API Transformation Process**. It also includes several supporting classes used internally, such as factories, utilities and helpers, validators, message repositories, and others.
  Because it is intended for internal use only, this package should not be visible to the API client, except for the *HappyTree* class, which serves as the entry point to the API.

Below are the remaining packages.

- **com.madzera.happytree.annotation**

  This package contains only the annotations used in the **API Transformation Process**. These annotations define the identifier, the parent, and the object's own class (the annotated class) that will be transformed into a node by the HappyTree API. This package is public, and the annotations provided are:

  - @Tree.
  - @Id.
  - @Parent.

- **com.madzera.happytree.exception**

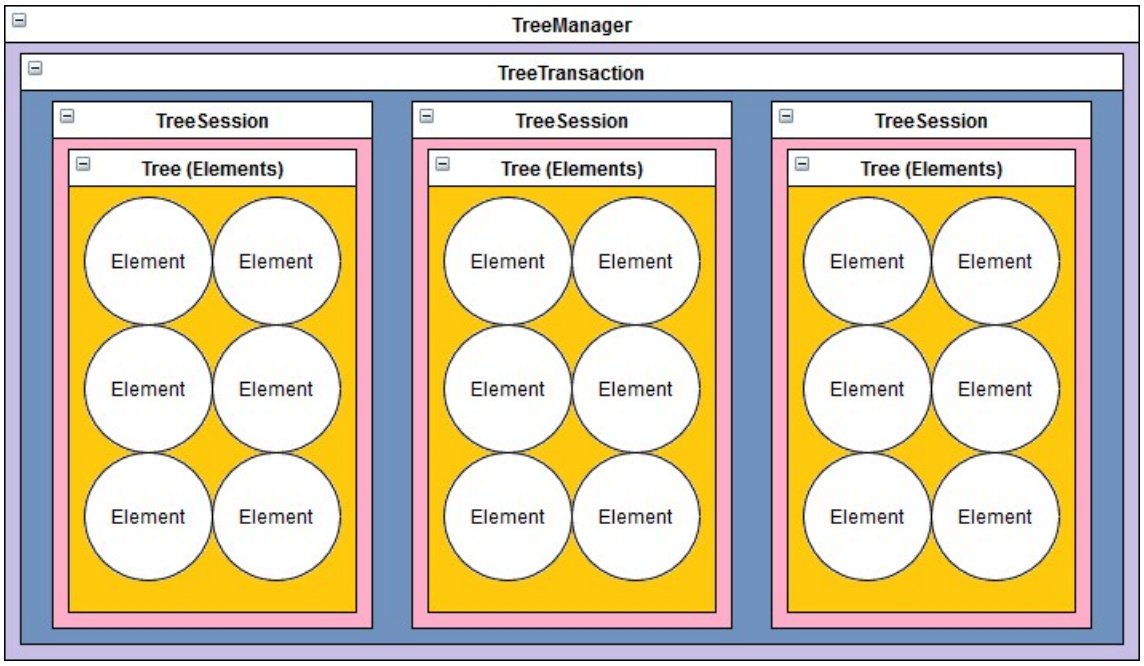
  This package contains the TreeException class, which is thrown whenever an error occurs. This package is public, as the API client is expected to handle this exception.

Structural Composition

An object of type Element represents a node in a tree. A tree can only exist within a previously initialized session (TreeSession). To initialize a session, the API client must invoke an object that represents a session transaction, known as a TreeTransaction.

However, a transaction can only be obtained through a manager that implements the TreeManager interface, which is provided to the API client.

In summary, every Element is inserted into a TreeSession, which is managed within a TreeTransaction, and ultimately accessed through a TreeManager.



Behavioral Composition

The HappyTree API allows the API client to interact with it in two distinct ways: manipulating elements within a tree and manipulating sessions. Each session owns its own tree, and no session can access or modify the trees of other sessions.

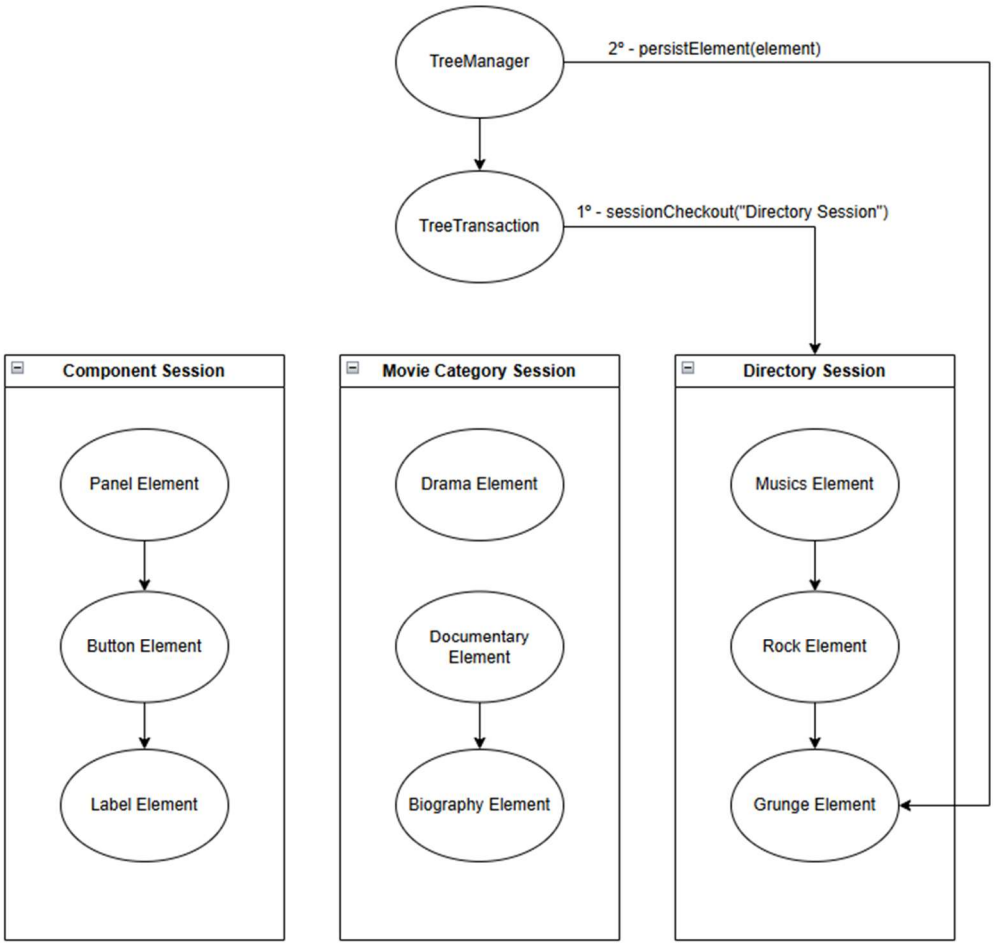
To enforce this isolation, a session mechanism was introduced. Within a session, each element has a unique identifier inside the tree, and each session itself has a unique identifier among all open sessions in a transaction.

As a result, the relationship between Element and TreeSession objects is intrinsic. The other two objects available to the API client, TreeManager and TreeTransaction, are responsible for manipulating Element and TreeSession objects, respectively.

While the TreeManager interface handles elements within sessions, the TreeTransaction instance—associated with the manager—acts as a cursor, selecting the session (tree) on which the TreeManager will operate.

The session state is a critical concern. The current session must always be active to allow tree manipulation; this is the first validation performed by the HappyTree API. Although multiple sessions can be active simultaneously, a TreeTransaction can operate on only one session at a time. The desired session is selected by the API client by invoking the *"transaction.sessionCheckout(sessionIdentifier)"* method.

Thus, there are two primary entities (Element and TreeSession) and two control entities (TreeManager and TreeTransaction) responsible for managing them.



In the image above, the TreeManager instance uses the transaction as a selector by first invoking *"`transaction.sessionCheckout(sessionIdentifier)`"* before executing an operation that directly affects the tree—represented in the image by the *"manager.persist(element)"* operation.

From the moment the API client selects a session through the *"`transaction.sessionCheckout(sessionIdentifier)`"* method, all operations performed by the TreeManager instance apply to the selected tree. If the transaction does not reference any session, or references an inactive session, a TreeException is thrown for all TreeManager operations.

Technical Details

Now that the main interfaces, their functionalities, their relationships, and their structural and behavioral compositions have been presented, it is possible to explore the technical details in greater depth.

The discussion begins with an explanation of the context, followed by a description of the phases involved in using the HappyTree API. Once these concepts are established, the lifecycle of Element objects is examined.

This initial explanation is essential for understanding why exceptions may be thrown, as the lifecycle of an Element within the tree forms the fundamental basis for fully utilizing the HappyTree API.

Subsequently, the session states are described, along with the specifications and required validations performed by the HappyTree API. To conclude, the **API Transformation Process** is explained.

Contexts

The HappyTree API is intended to manage object trees; however, it has no responsibility for the changes you make to these objects, which represent the nodes in the tree.

Consider the code below:

```
TreeManager manager = HappyTree.createTreeManager();
TreeTransaction transaction = manager.getTransaction();

Collection<Directory> directories = someObject.getDirectoryTree();

transaction.initializeSession("DirectoryTree", directories);

Element<Directory> winamp = manager.getElementById(winampId);
Element<Directory> programFiles = manager.getElementById(programFilesId);

programFiles.addChild(winamp);

//True or False?
manager.containsElement(programFiles, winamp);
```

Is the return of the last line ***true*** or ***false***?

Does the “*programFiles*” directory really have the “*winamp*” directory inside it, as a child, in the “*DirectoryTree*” session?

The answer is **false**. Although the "*programFiles*" object has the "*winamp*" object inside it, this change has not yet been synchronized in the "*DirectoryTree*" session. As previously stated, the HappyTree API has no responsibility for automatically synchronizing changes applied directly to tree objects.

When an element is retrieved from an already assembled tree, what is returned is a clone of the element. The actual instance of the element is never returned—only clones. Since the returned element may contain several children, they are all cloned and therefore represent identical copies of the elements that exist within the tree session.

There are two ways to complete the code above to move the "*winamp*" directory into *"programFiles"* within the “*DirectoryTree*” session:

"manager.updateElement(programFiles)"

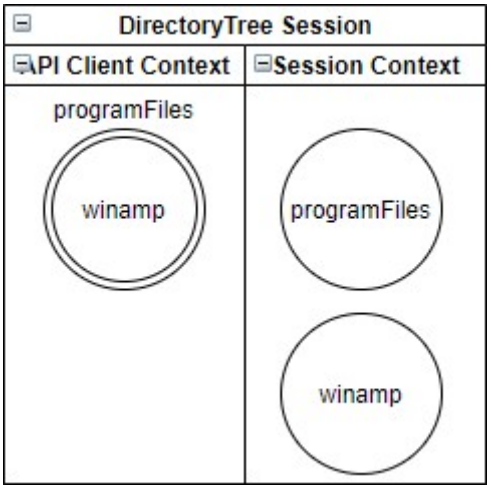
Or, alternatively, by invoking the method below without applying changes directly to the element:

"manager.cut(winamp, programFiles)"

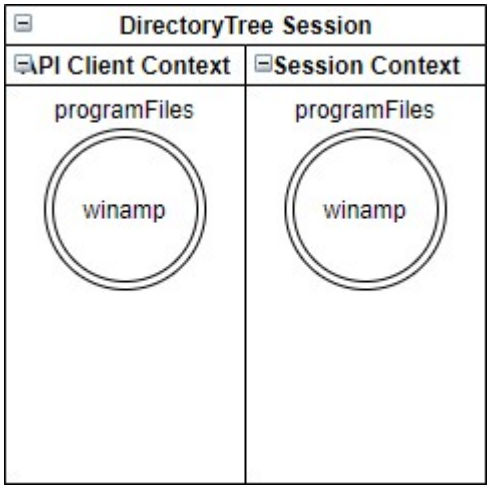
Note: when a tree change occurs through the TreeManager interface, it is not necessary to update the element. **Every change made via TreeManager is automatically synchronized with the tree**.

Based on everything discussed so far, there are two contexts, which can be understood as perspectives: the API client perspective and the session (the tree) perspective. Therefore, in relation to the example above, the following applies:

- Before synchronization



- After synchronization



Objects References

After synchronization, in the example above, it is necessary to ensure that the variables "*programFiles*" and "*winamp*" have their references updated, so they do not reference the state prior to synchronization.

```
TreeManager manager = HappyTree.createTreeManager();
TreeTransaction transaction = manager.getTransaction();

Collection<Directory> directories = someObject.getDirectoryTree();

transaction.initializeSession("DirectoryTree", directories);

Element<Directory> winamp = manager.getElementById(winampId);
Element<Directory> programFiles = manager.getElementById(programFilesId);

programFiles.addChild(winamp);

manager.updateElement(programFiles);

/*
* Still false at this point, despite the update. It is necessary to update the
* programFiles and winamp references.
*/
manager.containsElement(programFiles, winamp);

winamp = manager.getElementById(winampId);
programFiles = manager.getElementById(programFilesId);

//Now it is true.
manager.containsElement(programFiles, winamp);
```

The API client also needs to be especially careful with the immediate return of methods:

```
TreeManager manager = HappyTree.createTreeManager();
TreeTransaction transaction = manager.getTransaction();

Collection<Directory> directories = someObject.getDirectoryTree();

transaction.initializeSession("DirectoryTree", directories);

Element<Directory> winamp = manager.getElementById(winampId);
Element<Directory> programFiles = manager.getElementById(programFilesId);

programFiles.addChild(winamp);

manager.updateElement(programFiles);

/*
 * Still false because it invokes the containsElement(Object, Object)
 * method instead of containsElement(Element, Element).
 */
manager.containsElement(manager.getElementById(programFilesId),
              manager.getElementById(winampId));
```

In the example above, the API client intended to invoke the cut(Element, Element) method but instead invoked cut(Object, Object). This is another overload of the cut() method that accepts Object parameters instead of Element, representing the elements' @Id.

This occurs because the getElementById() method returns immediately within the containsElement() method. Since the HappyTree API relies on Java reflection, the JVM associates the return value directly with an Object instance at runtime.

Therefore, it is recommended to assign the return value of the method to a variable rather than using an immediate return.

Phases

Now that the concept of contexts in the HappyTree API has been explained, it is much easier to identify the execution phases. The following description applies equally to a new tree created from scratch or to a tree built through the **API Transformation Process**, as these phases are considered only after the session has been initialized.

There are three stages of execution. These stages have no direct impact on API usage and serve purely as an informational aid to facilitate understanding of the lifecycle of Element objects within sessions.

Phase	Method	Description
Initial Phase	`getElementById()` `search()`	Occurs when the API client retrieves an element. The returned element has not yet undergone any changes made by the API client.

Usage Phase	`createElement()`
	`addChild()`
	`addChildren()`
	`apply()`
	`setId()`
	`setParent()`
	`removeChild()`
	`removeChildren()`
	`wrap()`
		Occurs when the API client applies changes to the element state returned from the previous phase.

Synchronization Phase	`persistElement()` `updateElement()`	For the changes made in the previous phase to take effect, they must be synchronized with the tree session using the indicated methods. After synchronization, both contexts are aligned.

Element Lifecycle

The concepts of contexts and phases are presented here to provide a better understanding of the lifecycle of elements in the HappyTree API. Contexts depend on the element lifecycle to determine whether an element is synchronized with the actual tree, while the element lifecycle itself depends on the phase—specifically, the Usage Phase—for the HappyTree API to later determine whether the element was modified.

By using one of the methods specified in the Usage Phase (as described in the table above), the element changes its state. Consequently, when a method from the Synchronization Phase is invoked, the HappyTree API can detect that the element has been modified.

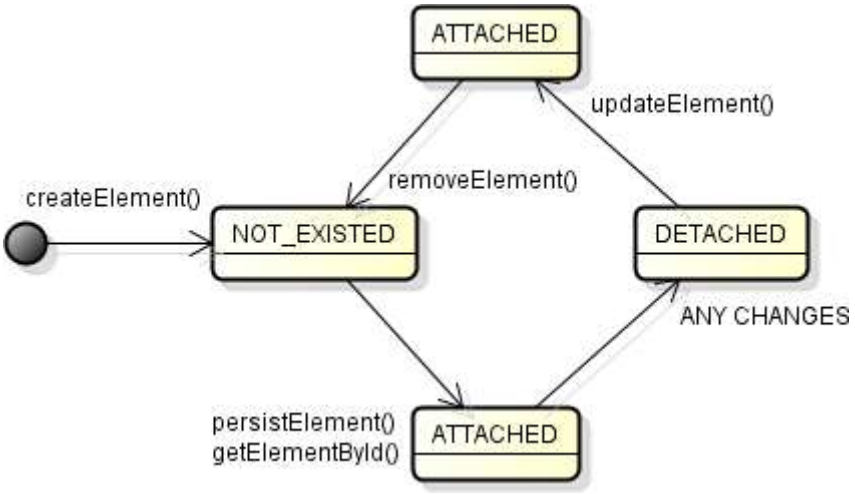
When an element is obtained from a tree session, the API client receives an **ATTACHED** element. This element represents an exact and faithful copy of the element—and all its descendants—relative to the session context, that is, the API client's own tree.

When the API client modifies the obtained element, the copy that was previously identical to the element in the session context no longer remains synchronized. This state is referred to as **DETACHED**, since the element is no longer synchronized with the tree.

Finally, when the API client determines that the element is no longer useful to the current tree, the element may be removed. When an element is removed from the tree within the context of a tree session, it is said to be in the **NOT_EXISTED** state. Similarly, when an element is created from scratch using "`manager.createElement(objectId, objectParentId, object)`", the API client creates an element that does not yet belong to the tree, and its state is therefore **NOT_EXISTED**.

It is important to note that all the explanations above apply only from the element's perspective. When using a writer method from the TreeManager interface, the modification is applied automatically, provided that all elements passed as parameters are in the **ATTACHED** state.

Thus, the lifecycle repeats itself from the moment an element is created or retrieved, until its eventual removal.



Sessions

The TreeSession interface is intended to represent a tree of elements. When referring to either the session or the tree, both concepts share the same meaning, since a session object contains the entire tree structure within it.

As mentioned at the beginning of this documentation, a transaction can operate on only one session at a time. To select the tree to work with, the API client must invoke the

"`transaction.sessionCheckout(sessionIdentifier)`" method. Alternatively, new sessions can be created by invoking either:

- `transaction.initializeSession("MyFirstHappyTree", Menu.class)`
- `transaction.initializeSession("My Tree Session ID", myObjects)`

This depends on the creation approach (from scratch or via the **API Transformation Process**, respectively).

Session State

The API client must ensure that the session it intends to handle is active; otherwise, a TreeException will be thrown. This validation is performed in almost all methods of the TreeManager object.

Basically, there are only three possible states for a session:

- **Activated**
  The session exists in memory and is enabled to be handled.
- **Deactivated**
  The session exists in memory but is not enabled to be handled. When a session is deactivated, the API client cannot manipulate the tree through TreeManager methods.
- **Destroyed**
  The session no longer exists in memory. In this case, the reference to the session object is null.

State	Exists?	Can it be handled?
Activated	✓	✓
Deactivated	✓	X
Destroyed	X	X

Session Initialization

The only way to create new sessions (new trees) is through the interface specifically responsible for session management, which is the TreeTransaction interface.

As mentioned, several times throughout this documentation, there are two ways to initialize (create) a session:

- **From scratch**

  In this approach, the API client creates a tree manually by creating each element individually and persisting them one by one.

  It is important to note that, in this case, the API client is responsible for providing all the information required to build each element, including:

- The element identifier.
- The parent element identifier.
- the object to be encapsulated within the element itself (wrapped object node).

- **From API Transformation Process**

  In this approach, the API client already has a pre-existing collection of objects annotated with @Tree, @Id, and @Parent, allowing these objects to be automatically arranged into a tree structure.

  In this case, all element information is assigned automatically during the **API Transformation Process**, including:

  - The element identifier.
  - The parent element identifier
  - The wrapped object node.

In both approaches, two parameters are mandatory when starting a new session:

- **Session Identifier**

  A string representing the session name. This parameter is required, and no other session may exist with the same name, including deactivated sessions.

- **Session Type**

  This parameter indicates the node class type of the tree (e.g., categories, directories, classifications, etc.).

  When initializing a session from scratch, the class type is passed explicitly, as shown in the example above (*"Menu.class"*).

  When initializing a session using the **API Transformation Process**, this type is implicitly defined by the collection passed as a parameter. As a result, each **wrapped object node** in the collection is automatically encapsulated within its corresponding element.

After initialization, the transaction automatically references the newly created session, regardless of whether the transaction was referencing a different session before the initialization process.

Multiple Session Management

When a transaction manages multiple sessions of the same or different types, the HappyTree API supports relocating elements between sessions through the TreeManager interface.

To perform this operation, it is important to understand certain aspects of session types, as well as the distinction between an element identifier and a session identifier:

- A session cannot have the same identifier as another session managed by the transaction, regardless of the session type.
- An element cannot share the same identifier value with another element within the same tree (i.e., the same session).
- An element may share the same identifier value with an element in a different tree, regardless of the type of that tree.

With these concepts in mind, when relocating an element to another tree—either by copying or moving it—using the methods respectively:

- `manager.copy(sourceElement, targetElement)`
- `manager.cut(sourceElement, targetElement)`

A TreeException will be thrown under the following conditions:

- The API client has not selected any session (active).
- The current session to which the "*sourceElement*" belongs, or the session to which the "*targetElement*" belongs, is not active.
- The "*sourceElement*" does not belong to the current session referenced by the transaction. This occurs when the current session differs from the session of the "*sourceElement*".
- The "*sourceElement*" is a root element.
- The "*sourceElement*", the "*targetElement*", or at least one of their child elements is in a **DETACHED** or **NOT_EXISTED** lifecycle state.
- The session containing the "*targetElement*" already has an element with the same **@Id** value as the "*sourceElement*".
- The session containing the "targetElement" has a different type from the session to which the "*sourceElement*" belongs.

Cloning Session

The HappyTree API provides built-in support for cloning trees. With a single method invocation, an entire tree can be cloned, with all its elements preserving the information from the original tree.

To clone a tree, the API client must invoke the "`transaction.cloneSession(from,to)`" method, where "from" can be either a TreeSession instance or a String representing the identifier of the source session, and "*to*" is a String representing the identifier of the newly cloned session.

Note: If a session already exists with the same identifier as the "*to*" parameter, this method will overwrite the existing tree, resulting in the complete loss of any previously

stored information. It is the developer's responsibility to verify that the target session identifier is not already in use before invoking this method**.**

Specifications & Validations

The HappyTree API performs a series of validations to prevent inconsistencies that violate its specifications. These validations occur in two situations: during the **API Transformation Process**, and when invoking methods of the TreeManager interface after the tree has been built.

The HappyTree API may throw exceptions of two types: TreeException and IllegalArgumentException.

TreeException is an exception class specific to the HappyTree API and is thrown whenever an API specification is violated.

IllegalArgumentException is a runtime exception native to Java. Within the context of the HappyTree API, this exception is thrown when input parameters are null.

TreeTransaction - API Transformation Process (before the tree is built)		
Specification	**Message**	**Type**
The input parameters must not be null.	Invalid null/empty argument(s).	IllegalArgumentException
The session identifier must be unique.	Duplicate session identifier.	TreeException
The class of the object to be transformed must be annotated with **@Tree**.	No @Tree annotation found.	TreeException
The identifier of the object to be transformed must be annotated with **@Id**.	No @Id annotation found.	TreeException
The parent identifier of the object to be transformed must be annotated with **@Parent**.	No @Parent annotation found.	TreeException
The class of the object to be transformed must have a default constructor, *getters*, and *setters*.	Unable to transform input objects. Ensure the presence of a default constructor, getters, and setters.	TreeException
The value of the **@Id** attribute must not be null.	Invalid null/empty argument(s).	IllegalArgumentException
The value of the **@Id** attribute must be unique within the same tree session.	Duplicate ID.	TreeException
The @Id and @Parent attributes must be of the	ID type mismatch error.	TreeException

same type.		
The wrapped object node does not implement the Serializable interface.	The wrapped object must implement Serializable.	TreeException

TreeManager - Methods (after the tree is built)		
Specification	**Message**	**Type**
The input parameters must not be null.	Invalid null/empty argument(s).	IllegalArgumentException
When invoking an operation that directly handles elements in the tree. The transaction must refer to a defined session.	No defined session.	TreeException
When invoking an operation that directly handles elements in the tree, the transaction must refer to an active session.	No active session.	TreeException
When handling an element, ensure that the associated transaction references the session to which the element belongs.	Element not defined in this session.	TreeException
When copying or moving an element from one tree to another, both trees must have the same type of object that the element wraps.	Type mismatch error: incompatible parameterized tree type.	TreeException
It is not possible to perform operations on elements that represent the root of a tree.	The root of the tree cannot be handled for this operation.	TreeException
Operations that change the state of the tree can only be performed depending on the lifecycle of the elements involved in these operations.	• It is not possible to copy/cut/remove elements. Invalid lifecycle state. • It is not possible to persist the element. Invalid lifecycle state. • It is not possible	TreeException

	to update the element. Invalid lifecycle state.	
Duplicate ID elements are not allowed within the same tree.	Duplicate ID.	TreeException
When attempting to cut an element by its **@Id**, the operation fails if the element does not exist.	It is not possible to cut the element. Source element not found.	TreeException

API Transformation Process

As mentioned earlier, this mechanism is responsible for transforming a linear structure of Java model objects that logically exhibit tree-like behavior but are not structurally represented as such.

The expression *"having a tree behavior even though it is not structurally represented as one"* refers to a collection of objects that are logically related to one another, where one object is the child of another, but where these relationships are not expressed through structural containment.

This mechanism therefore transforms the linear structure so that objects become structurally nested within one another. As a result, an object may contain a list of child objects, each of which may, in turn, contain its own list of children, and so on.

As previously discussed, there are two ways to initialize a session. One of them involves passing a collection of objects to be transformed, which triggers the **API Transformation Process**. Let us now review this process in more detail.

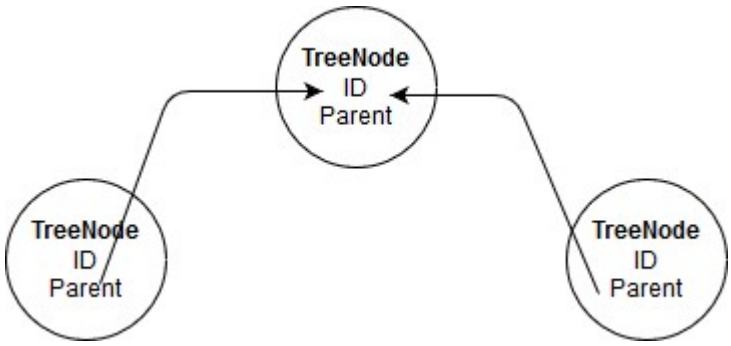
Creating a new tree from scratch

In this case, no **API Transformation Process** is involved. The API client simply initializes a standard new tree session to be handled afterward. As a result, the tree initially contains only the root element.

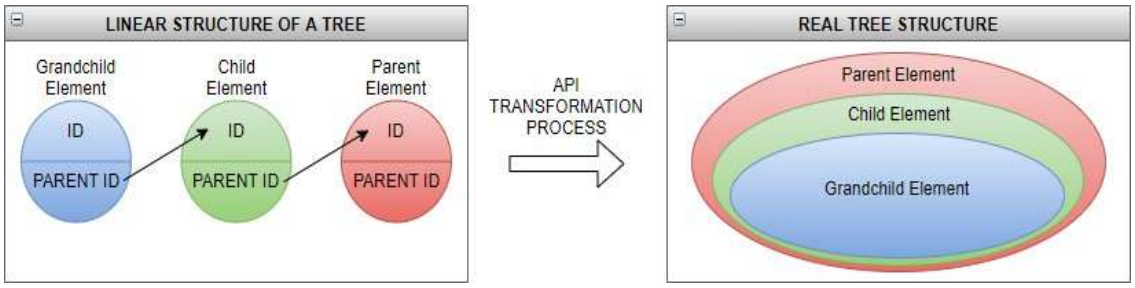
The method used to initialize a standard tree session is Transaction.initializeSession(String, Class), where the String parameter represents the session identifier (which must be unique and not null), and the Class parameter represents the parameterized type of the tree. This type is used by the Element interface to wrap the object that represents a node in the tree.

Creating a new tree using the API

The API client has a structure that represents a tree, but it is designed in a linear form, through the TreeNode example class as shown below:



This structure is then transformed into:



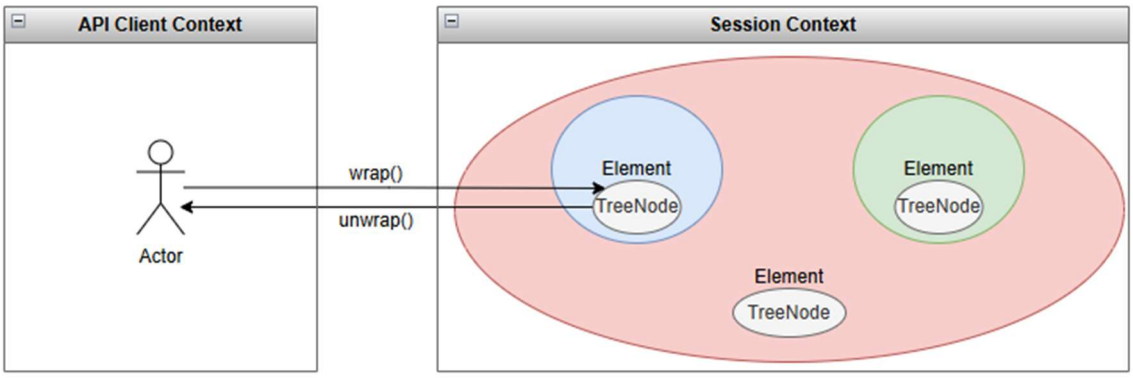
The version of the method to initialize a tree session through **API Transformation Process** is Transaction.initializeSession(String, Collection) where String is the session identifier (unique and not null) and Collection represents the list of objects to be transformed by the **API Transformation Process**.

This collection contains objects that their class is annotated by @Tree, @Id and @Parent and consequently represents the parameterized type of the tree. During the transformation process (**API Transformation Process** lifecycle), these objects will be automatically wrapped within their respective elements in the current tree session, thus representing nodes in the tree. To unwrap the respective object from an element, simply invoke Element.unwrap().

Note that the original object is now encapsulated within an Element. Therefore, when the API client wants to retrieve the object corresponding to a specific position in the tree, it must first locate the corresponding element (for example, by using Element.getElementById(Object)) and then extract the object by invoking Element.unwrap().

When extracting the **wrapped object node** using Element.unwrap(), the HappyTree API always returns a copy of the original object. If the API client needs to modify any property of this object, it must invoke Element.wrap(TreeNode) to re-wrap the modified object

and then call TreeManager.updateElement(Element) to synchronize and apply the change.

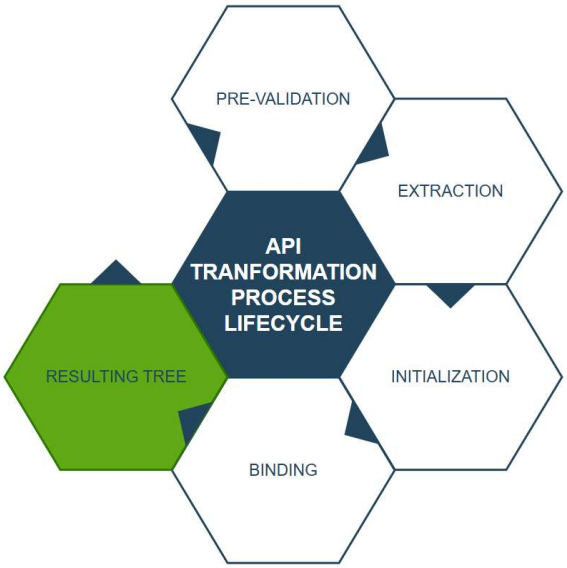


ATP Lifecycle

The **API Transformation Process** has an internal lifecycle composed of distinct phases that aim to transform a linear structure of objects—which logically represent a tree data structure—into an actual tree.

This lifecycle has no functional impact on how the API client uses the **API Transformation Process**. The explanation provided here is purely informational and is intended to help users better understand how a tree is assembled from a legacy linear structure of objects that logically represent hierarchical relationships.

As input, the **API Transformation Process** receives a collection of objects that will be transformed into a tree through five distinct and consecutive phases:



1. Pre-Validation

This phase performs a set of validations to verify whether the received input complies with the adopted specifications. An IllegalArgumentException may be thrown if the list

of objects to be transformed is null or empty, while a TreeException is thrown when any other specification is violated.

The following validations are performed:

- Verifies that the list of objects to be transformed is not null or empty.
- Verifies whether a session with the same identifier already exists.
- Verifies that the class of the objects to be transformed is annotated with @Tree.
- Verifies that the class of the objects to be transformed is annotated with @Id.
- Verifies that the class of the objects to be transformed is annotated with @Parent.
- Verifies that the @Id and @Parent attributes have the same type.
- Verifies whether the class of the objects implements Serializable.
- Verifies whether any object has a null value for the @Id attribute.
- Checks for duplicate @Id values.
- Verifies that the class of the objects to be transformed provides valid *getters* and *setters*.

2. Extraction

If the input list of objects passes all validations from the previous phase, the HappyTree API proceeds to extract the objects to separate them from their respective parent references. As a result, the output of this phase consists of two distinct groups: the source objects and their corresponding parent references.

3. Initialization

In this phase, the HappyTree API instantiates an Element object for each source object provided as input. The values of the @Id and @Parent attributes from the source object are assigned to the corresponding element.

In addition, the source object itself is automatically wrapped within the element, making it eligible to represent a node in the tree, since an Element naturally represents a node in the context of the HappyTree API.

After the tree has been built, the original source object can be retrieved by invoking the Element.unwrap() method.

As a result of this phase, all elements are instantiated and contain the complete information derived from the source objects.

4. Binding

After obtaining the list of elements from the previous phase, the HappyTree API binds each element to its respective parent using the parent information extracted during the Extraction phase.

This is the phase in which the tree is actually assembled. For each node in the tree, there is a corresponding Element object, where each element contains:

- The value of the @Id attribute.
- The value of the @Parent attribute.
- The **wrapped object node** corresponding to the source object transformed during the process.
- A collection of child elements representing its direct descendants.
- The tree session to which the element belongs.