# New Features in Ice 3.7

This page describes notable additions and improvements in Ice 3.7. For a detailed list of the changes in this release, please refer to the changelog in the source tree. Our upgrade guide documents the changes that may affect the operation of your applications or have an impact on your source code.

On this page:



# Main New Features

This section describes the major new features included in Ice 3.7.

## Ice-E and Ice Touch Merged into Ice

Ice-E and Ice Touch are now part of Ice. They are no longer separate products.

In particular, the latest Ice binary distribution for macOS includes support for Xcode SDKs and iOS. Refer to Using the macOS binary distribution for information on how to use the Xcode SDKs provided with the Ice binary distribution.

Back to Top ^

## New C++11 Mapping

Ice now provides two distinct Slice to C++ mappings:

- The C++98 mapping, which corresponds to the C++ mapping provided by prior Ice releases.
- A new C++11 mapping, which takes full advantage of C++11 features, including standard smart pointers, move semantics, lambda expressions, futures and promises, and much more.

We provide an overview of some of the new features in the C++11 mapping below. Please refer to the C++11 Mapping and C++98 Mapping chapters for additional information.

### Standard shared_ptr for Everything

With the C++11 mapping, virtually all Ice objects are manipulated through `std::shared_ptr` smart pointers. For example:

**C++**

```
// ich.communicator() is a std::shared_ptr<Ice::Communicator>
// oa is a std::shared_ptr<Ice::ObjectAdapter>
//
Ice::CommunicatorHolder ich(argc, argv);
auto oa = ich.communicator()->createObjectAdapter("Hello");

// servant is a std::shared_ptr<HelloI>
// proxy is a std::shared_ptr<Ice::ObjectPrx>
//
auto servant = make_shared<HelloI>();
auto proxy = oa->addWithUUID(servant);
oa->activate();

// address is a std::shared<Address> (Address is a mapped Slice class)
//
auto address = make_shared<Address>();
person->address = address;
```

## AMI

With the C++11 mapping, Ice provides two options for asynchronous method invocation (AMI): a function that returns a `std::future` of the result, and a function that takes `std::function` "callbacks" to process the result. For example, the Slice operation `string getName()` is mapped to a proxy class with the following Async functions:

**C++**

```
std::future<std::string> getNameAsync(const Ice::Context& context = Ice::noExplicitContext);

std::function<void()>
getNameAsync(std::function<void(std::string)> response,
             std::function<void(std::exception_ptr)> ex = nullptr,
             std::function<void(bool)> sent = nullptr,
             const Ice::Context& context = Ice::noExplicitContext);
```

Your code can then use the standard future API, or lambda expressions, to process the result of these asynchronous calls. For example:

**C++ with std::future**

```
auto fut = proxy->getNameAsync();

try
{
    cout << fut.get() << endl;
}
catch(const std::exception& ex)
{
   // something went wrong...
}
```

or

**C++ with callbacks**

```
proxy->getNameAsync([](string name) { cout << name << endl; },
                    [](std::exception_ptr eptr) { ... deal with error... });
```

## AMD

With the C++11 mapping, the API for asynchronous method dispatch (AMD) closely resembles its AMI counterpart with callbacks:

**C++**

```
 virtual void getNameAsync(std::function<void(const std::string&)> response,
                           std::function<void(std::exception_ptr)> ex,
                           const Ice::Current&) = 0;
```

Even though the response callbacks are not identical in AMI and AMD, they are compatible, which allows you to easily chain AMD and AMI calls:

**C++**

```
// AMD implementation of Person::getName
void
PersonI::getNameAsync(std::function<void(const std::string&)> response,
                      std::function<void(std::exception_ptr)> ex,
                      const Ice::Current&)
{
     // Call the same operation on _proxy with AMI
    _proxy->getNameAsync(response, ex);
}
```

## Movable Parameters

On the server-side, the generated code allocates `in` parameters on the stack before dispatching the call to your operation implementation. With the C++98 mapping, "big" parameters such as strings, structs and sequences are passed as `const&`, just like on the client-side. With the C++11 mapping, they are passed as values, which allows you to move them. For example:

**Slice**

```
interface TextTransfer
{
    void sendText(string longText);
}
```

is mapped to:

**C++**

```
// Server-side
virtual void sendText(std::string, const Ice::Current&) = 0; // std::string, not const std::string&
```

So you can now keep this parameter without additional memory allocation:

**C++**

```
// Implement sendText
void
TestTransferI::sendText(std::string longText, const Ice::Current&)
{
    std::lock_guard<std::mutex> lk(_mutex);
    _text = std::move(longText); // move-assignment
}
```

The same rule applies for any parameter that Ice allocates and then gives to your application, like (for example) return and out parameters given to AMI response callbacks.

Back to Top ^

# New Java Mapping

Similar to what we've done for C++, Ice 3.7 supports two Java mappings: Java and Java Compat. As its name suggests, the Java Compat mapping is provided primarily for backward compatibility purposes so that existing Java applications can be upgraded to Ice 3.7 without requiring much change. Note however that the Java Compat mapping will be removed in the next release and we recommend migrating applications to the new Java mapping as soon as possible.

The primary goals of the new Java mapping were modernization, standardization, and simplification. We provide an overview of the new features below. Please refer to the Java Mapping and Java Compat Mapping chapters for additional information.

Regardless of whether you use the Java Compat mapping or the Java mapping, the minimum required Java version for Ice 3.7 is Java 8.

## AMI

The API for asynchronous method invocation (AMI) now uses Java's `CompletableFuture` class. For example, the Slice operation `string getName()` would be mapped as follows:

**Java**

```
java.util.concurrent.CompletableFuture<String> getNameAsync()
```

Clients can easily use lambdas as asynchronous callbacks by configuring the future:

**Java**

```
proxy.getNameAsync().whenComplete((name, ex) ->
    {
        if(ex != null)
        {
            // oops!
        }
        else
        {
            System.out.println("got name " + name);
        }
    });
```

The `CompletableFuture` class provides a great deal of flexibility and power for implementing your asynchronous programming requirements.

## AMD

As with AMI, the API for asynchronous method dispatch (AMD) has also changed significantly. Now the mapping closely resembles its AMI counterpart in that an AMD operation returns a `CompletionStage`:

**Java**

```
java.util.concurrent.CompletionStage<String> getNameAsync(com.zeroc.Ice.Current current)
```

The implementation is responsible for creating and returning a completion stage that must eventually be completed with a result or an exception. Typically the implementation will return an instance of `CompletableFuture`, which implements the `CompletionStage` interface.

## Out Parameters

The Java mapping no longer uses "holder" classes to provide the values of out parameters. For Slice operations that return a single value (either as the return type or as an out parameter), the mapped method provides the value as its return type. Consider these operations:

**Slice**

```
interface Example
{
    string getNameRet();
    void getNameOut(out string s);
}
```

The mapping for these two operations is identical:

**Java**

```
public interface ExamplePrx ...
{
    String getNameRet();
    String getNameOut();
}
```

For Slice operations that return multiple values, the mapping generates an extra class to hold the operation's results:

**Slice**

```
interface Example
{
    string getAll(out int count);
}
```

The mapping for `getAll` is shown below:

**Java**

```
public interface Example ...
{
    public static class GetAllResult
    {
        ...
        public String returnValue;
        public int count;
    }
    ...
}

public interface ExamplePrx ...
{
    Example.GetAllResult getAll();
}
```

Changing the API to ensure that all mapped operations return at most one logical value enabled us to use `CompletableFuture` and `CompletionStage` in our AMI and AMD mappings, respectively.

### Optional Values

As part of the standardization goal, optional parameters and data members are now mapped to the `java.util.Optional` family of classes.

### Servant Interfaces and Tie Classes

Tie classes are no longer necessary or supported with the Java mapping. Tie classes were a useful implementation technique in prior Ice versions that allowed a servant to implement Slice operations without the need to extend a generated class, making it possible for the servant to extend an arbitrary base class.

Now all servant code is generated in an interface, so your servant class only needs to implement the generated interface and can extend an arbitrary base class without the need for a tie class.

### Packaging

All classes have been moved to the `com.zeroc` package (e.g., `com.zeroc.Ice.Communicator`).

Back to Top ^

## C# Support for async and await

The C# mappings for asynchronous method invocation (AMI) and asynchronous method dispatch (AMD) are completely new in Ice 3.7 and take advantage of the new asynchronous programming features introduced with .NET 4.5.

The `begin_/end_` proxy API from previous Ice versions is still supported for backward compatibility purposes but is now deprecated and will be removed in the next Ice version. No backward-compatible API is provided for AMD operations.

The sections below provide an overview of these changes.

## AMI

The API for asynchronous method invocation (AMI) now uses .NET's `Task` class. For example, the Slice operation `string getName()` would be mapped as follows:

**C#**

```
System.Threading.Tasks.Task<string> getNameAsync()
```

Clients can use the `await` keyword to suspend processing of the current thread and resume it as a continuation when the task is complete:

**C#**

```
var name = await proxy.getNameAsync();

// use the return value in a continuation...
```

You could also interact with the task directly:

**C#**

```
var task = proxy.getNameAsync();

// do something else

var name = task.Result; // blocks until the result is available
```

Or configure a lambda to execute as the continuation:

**C#**

```
proxy.getNameAsync().ContinueWith((name) =>
    {
        Console.WriteLine("got name " + name);
    });
```

## AMD

As with AMI, the API for asynchronous method dispatch (AMD) has also changed significantly. Now the mapping closely resembles its AMI counterpart in that an AMD operation returns a `Task`:

**C#**

```
System.Threading.Tasks.Task<string> getNameAsync(Ice.Current current)
```

The implementation is responsible for creating and returning a task that must eventually be completed with a result or an exception.

## Out Parameters

The C# mapping for AMI and AMD no longer uses out parameters. For Slice operations that return a single value (either as the return type or as an out parameter), the mapped method provides the value as its return type. Consider these operations:

**Slice**

```
interface Example
{
    string getNameRet();
    void getNameOut(out string s);
}
```

The AMI mapping for these two operations is identical:

**C#**

```
public interface ExamplePrx ...
{
    Task<string> getNameRetAsync();
    Task<string> getNameOutAsync();
}
```

For Slice operations that return multiple values, the mapping generates an extra type to hold the operation's results:

**Slice**

```
interface Example
{
    string getAll(out int count);
}
```

The mapping for `getAll` is shown below:

**C#**

```
public struct Example_GetAllResult
{
    ...
    public string returnValue;
    public int count;
}

public interface ExamplePrx ...
{
    Task<Example_GetAllResult> getAllAsync();
}
```

Changing the API to ensure that all mapped operations return at most one logical value enabled us to use tasks in our AMI and AMD mappings, respectively.

Back to Top ^

# JavaScript

## JavaScript 6 Updates

The JavaScript mapping has been updated to take advantage of the latest features present in the EcmaScript 6 standard:

- All Promise objects returned by the Ice run time are derived from the standard Promise type.
- The mapping for dictionaries is now based on the standard Map type. The `Ice.HashMap` type used in previous releases is still supported for a few cases that are not covered by the standard Map type. See JavaScript mapping for dictionaries for complete details.
- There is a new mapping for Slice modules based on the new JavaScript import and export keywords. See JavaScript mapping for modules for more details.

## JavaScript Changes for AMD

The API for Asynchronous Method Dispatch (AMD) has been updated to use the standard Promise class. There are several differences with respect to the previous API:

- The `["amd"]` metadata is ignored by the Slice to JavaScript compiler.
- The Servant skeleton does not generate separate `_async` methods for AMD.
- At run time a servant can take advantage of AMD by returning a Promise object.

Consider the following example, where we define a `getName` operation in Slice:

```
module M
{
    interface Example
    {
        string getName();
    }
}
```

The JavaScript implementation can decide at run time to use the synchronous or asynchronous dispatch. If the servant returns a Promise, the Ice run time uses asynchronous dispatch and marshals the result when the promise is resolved. Otherwise, Ice uses synchronous dispatch and marshals the result right away. Here's an example that demonstrates the mapping:

```
class ServantI extends Servant
{
    getName(current)
    {
        if(_name !== undefined)
        {
            return _name; // Use synchronous dispatch
        }
        else
        {
            // Use asynchronous dispatch.
                        // All JavaSript proxy invocations return a standard Promise object.
            // We can directly return the promise returned by getName from our
            // servant implementation.
            return _proxy.getName();
        }
    }
}
```

Back to Top ^

## Python Changes for AMI and AMD

We've added a new AMI mapping and modified the AMD mapping in Python.

The existing AMI mapping is still supported for backward compatibility, but new applications should use the new AMI mapping. The changes to the AMD mapping break backward compatibility and require modifications to existing applications. Refer to the upgrade guide for more information.

### AMI

The new API for asynchronous method invocation (AMI) uses future objects and adds the `Async` suffix to proxy methods. For example, the Slice operation `string getName()` would be mapped as follows:

| **Python** |
| --- |
| `def getNameAsync()` |

Asynchronous proxy methods return instances of `Ice.Future`, which has an API similar to the built-in Python types `asyncio.Future` and `concurrent.futures.Future`.

Clients can easily configure the future with a completion callback:

**Python**

```
def callback(future):
    try:
        print("got name " + future.result())
    except:
        # oops!

proxy.getNameAsync().add_done_callback(callback)
```

The `Ice.Future` class also provides methods for checking the status of an invocation, cancelling an invocation, and blocking until the invocation completes, among others.

### AMD

The API for asynchronous method dispatch (AMD) has changed significantly:

- The `_async` suffix is no longer used in the name of a dispatch method
- The dispatch method does not receive a callback object
- The dispatch method can either return results directly (like a synchronous dispatch method), or return a future object to be completed later

In essence, the synchronous and asynchronous mappings have been merged into a single mapping in which the value returned at run time dictates the semantics of the dispatch. If the implementation returns a future, Ice configures it with a callback and marshals the results when the future completes. For all other return types, Ice assumes the dispatch completed successfully and marshals the results immediately.

Here's an example:

**Python**

```
def getName(self, current=None):
    if self._name:
        return self._name # Name was cached, synchronous dispatch
    else:
        f = Ice.Future()  # We have to complete this future eventually
        ...
        return f          # Asynchronous dispatch
```

### Python 3.5 Features

The Python mapping adds the following features for applications using Python 3.5 or later:

- Servant dispatch methods can be implemented as coroutines
- `Ice.Future` objects are awaitable, meaning they can be targets of the `await` keyword

Aside from these features, all of the improvements to the Python mapping can be used in Python 2.x or later.

Back to Top ^

## New Database Back-end for IceGrid and IceStorm

IceGrid and IceStorm now store their data in LMDB databases. LMDB is a popular embedded database system known for its speed and reliability.

In prior Ice releases, IceGrid and IceStorm relied on Freeze for persistent storage, and Freeze itself stores its data in Oracle Berkeley DB databases. The migration instructions describe how to migrate your existing IceGrid and IceStorm databases to the new LMDB format.

Back to Top ^

## Bluetooth Transport for Linux and Android

Our newest transport plug-in enables Ice applications on Linux and Android to communicate with one another via Bluetooth. Once two devices are paired, establishing a peer-to-peer Bluetooth connection is just as convenient as with any other Ice transport. For example, a client can use a stringified proxy like this:

```
objectId:bt -a "01:23:45:67:89:AB" -u dfde1c02-d907-4ca1-bd99-31804c569624
```

The `-a` option denotes the device address of the peer and the `-u` option defines the unique identifier (UUID) associated with the desired Ice service on the peer device. It's also possible to secure the bluetooth connection with SSL by configuring IceSSL and using a `bts` endpoint instead.

Our ice-demos repository includes a new Android sample program that demonstrates how to use the transport. Refer to the Ice manual for more information on the plug-in.

Back to Top ^

## iAP Transport for iOS

The new Ice iAP transport enables iOS clients to communicate with accessories over Bluetooth or the Apple Lightning or 30-pin connector.

> This transport requires special support on the accessory-side and is only provided as a preview to demonstrate how it can be used in iOS applications to support Bluetooth communications. Please contact us for more information.

For example, a client can use a stringified proxy like this to communicate with an accessory that advertises the `com.zeroc.helloWorld` protocol:

```
objectId:iap -p com.zeroc.helloWorld
```

It's also possible to secure the accessory connection with SSL by configuring IceSSL and using an `iaps` endpoint instead.

Refer to the Ice manual for more information on the plug-in.

Back to Top ^

# Other New Features

This section describes other noteworthy features introduced with Ice 3.7.

## Optional Semicolons after Braces in Slice

Semicolons are now optional after a closing brace in Slice. For example, with Ice releases up to Ice 3.6, you would write:

**Slice**

```
module M
{
    enum E { A, B, C, D };

    interface Foo
    {
        void op();
    };
};
```

With Ice 3.7, the definitions above remain valid, but you can also remove the semicolons after the closing braces, as in:

**Slice**

```
module M
{
    enum E { A, B, C, D }

    interface Foo
    {
        void op();
    }
}
```

Back to Top ^

## Simplified Communicator Destruction

Ice 3.7 provides support for automatic communicator destruction in most programming languages.

- C++: `Ice::CommunicatorHolder` RAII Helper

- C#:  Communicator implements `IDisposable` (since Ice 3.6)
- Java: Communicator is `AutoCloseable`
- Python: Communicator implements the Python context manager protocol

Back to Top ^

## Freeze Unbundled

The Freeze persistent service is no longer included in the ice source repository or in the Ice binary distributions. It's now a separate add-on, with its own source repository and binary distributions.

Back to Top ^