C++ embedder API

Node.js provides a number of C++ APIs that can be used to execute JavaScript in a Node.js environment from other C++ software.

The documentation for these APIs can be found in src/node.h in the Node.js source tree. In addition to the APIs exposed by Node.js, some required concepts are provided by the V8 embedder API.

Because using Node.js as an embedded library is different from writing code that is executed by Node.js, breaking changes do not follow typical Node.js deprecation policy and may occur on each semver-major release without prior warning.

Example embedding application

The following sections will provide an overview over how to use these APIs to create an application from scratch that will perform the equivalent of `node -e <code>`, i.e. that will take a piece of JavaScript and run it in a Node.js-specific environment.

The full code can be found in the Node.js source tree.

Setting up per-process state

Node.js requires some per-process state management in order to run:

- Arguments parsing for Node.js CLI options,
- V8 per-process requirements, such as a `v8::Platform` instance.

The following example shows how these can be set up. Some class names are from the `node` and `v8` C++ namespaces, respectively.

```
int main(int argc, char** argv) {
  argv = uv_setup_args(argc, argv);
  std::vector<std::string> args(argv, argv + argc);
  std::vector<std::string> exec_args;
  std::vector<std::string> errors;
  // Parse Node.js CLI options, and print any errors that have occurred while
  // trying to parse them.
  int exit_code = node::InitializeNodeWithArgs(&args, &exec_args, &errors);
  for (const std::string& error : errors)
    fprintf(stderr, "%s: %s\n", args[0].c_str(), error.c_str());
  if (exit_code != 0) {
    return exit_code;
  }

  // Create a v8::Platform instance. `MultiIsolatePlatform::Create()` is a way
  // to create a v8::Platform instance that Node.js can use when creating
  // Worker threads. When no `MultiIsolatePlatform` instance is present,
  // Worker threads are disabled.
  std::unique_ptr<MultiIsolatePlatform> platform =
      MultiIsolatePlatform::Create(4);
  V8::InitializePlatform(platform.get());
  V8::Initialize();
```

```
  // See below for the contents of this function.
  int ret = RunNodeInstance(platform.get(), args, exec_args);

  V8::Dispose();
  V8::DisposePlatform();
  return ret;
}
```

Per-instance state

Node.js has a concept of a "Node.js instance", that is commonly being referred to as `node::Environment`. Each `node::Environment` is associated with:

- Exactly one `v8::Isolate`, i.e. one JS Engine instance,
- Exactly one `uv_loop_t`, i.e. one event loop, and
- A number of `v8::Context`s, but exactly one main `v8::Context`.
- One `node::IsolateData` instance that contains information that could be shared by multiple `node::Environment`s that use the same `v8::Isolate`. Currently, no testing if performed for this scenario.

In order to set up a `v8::Isolate`, an `v8::ArrayBuffer::Allocator` needs to be provided. One possible choice is the default Node.js allocator, which can be created through `node::ArrayBufferAllocator::Create()`. Using the Node.js allocator allows minor performance optimizations when addons use the Node.js C++ `Buffer` API, and is required in order to track `ArrayBuffer` memory in `process.memoryUsage()`.

Additionally, each `v8::Isolate` that is used for a Node.js instance needs to be registered and unregistered with the `MultiIsolatePlatform` instance, if one is being used, in order for the platform to know which event loop to use for tasks scheduled by the `v8::Isolate`.

The `node::NewIsolate()` helper function creates a `v8::Isolate`, sets it up with some Node.js-specific hooks (e.g. the Node.js error handler), and registers it with the platform automatically.

```
int RunNodeInstance(MultiIsolatePlatform* platform,
                    const std::vector<std::string>& args,
                    const std::vector<std::string>& exec_args) {
  int exit_code = 0;

  // Setup up a libuv event loop, v8::Isolate, and Node.js Environment.
  std::vector<std::string> errors;
  std::unique_ptr<CommonEnvironmentSetup> setup =
      CommonEnvironmentSetup::Create(platform, &errors, args, exec_args);
  if (!setup) {
    for (const std::string& err : errors)
      fprintf(stderr, "%s: %s\n", args[0].c_str(), err.c_str());
    return 1;
  }

  Isolate* isolate = setup->isolate();
  Environment* env = setup->env();
```

```
  {
    Locker locker(isolate);
    Isolate::Scope isolate_scope(isolate);
    // The v8::Context needs to be entered when node::CreateEnvironment() and
    // node::LoadEnvironment() are being called.
    Context::Scope context_scope(setup->context());

    // Set up the Node.js instance for execution, and run code inside of it.
    // There is also a variant that takes a callback and provides it with
    // the `require` and `process` objects, so that it can manually compile
    // and run scripts as needed.
    // The `require` function inside this script does *not* access the file
    // system, and can only load built-in Node.js modules.
    // `module.createRequire()` is being used to create one that is able to
    // load files from the disk, and uses the standard CommonJS file loader
    // instead of the internal-only `require` function.
    MaybeLocal<Value> loadenv_ret = node::LoadEnvironment(
        env,
        "const publicRequire ="
        "  require('module').createRequire(process.cwd() + '/');"
        "globalThis.require = publicRequire;"
        "require('vm').runInThisContext(process.argv[1]);");

    if (loadenv_ret.IsEmpty())  // There has been a JS exception.
      return 1;

    exit_code = node::SpinEventLoop(env).FromMaybe(1);

    // node::Stop() can be used to explicitly stop the event loop and keep
    // further JavaScript from running. It can be called from any thread,
    // and will act like worker.terminate() if called from another thread.
    node::Stop(env);
  }

  return exit_code;
}
```