DyNAMiC Workbench Architecture

Casey Grun

1 Overview

1.1 Introduction

This document describes the application architecture for the DyNAMiC Workbench IDE. The purpose of this document is to provide a high-level overview of the components which comprise the Workbench software client and server—and to explain how those components relate to one another—for a prospective developer/maintainer of the software. This does *not* attempt to document the application programming interface (API) for components of the software; API documentation is provided separately. Likewise, this is not "Help" documentation for a user; that is also provided elsewhere. Some basic familiarity with DNA nanotechnology and the functions of the Workbench software is presumed, but this document primarily describes components from a software perspective—that is, it explains their function in relation to other software components and to the larger system architecture, rather than their specific utility to a DNA nanotechnologist.

1.2 Relevant technologies

The Workbench application is written primarily in Javascript; this includes both the client and the server. Client-side Javascript is executed by a web browser. Server-side Javascript is executed by Node.js—a platform which provides Chrome's V8 Javascript engine with networking and file I/O capabilities. Client-side code uses HTML5, CSS, and SVG to provide a graphical interface, and Javascript to provide interactivity and networking (e.g. for communication with the server). JSON is used extensively for communication between the client and the server. Server-side code uses Javascript for networking and file I/O. Computationally-intensive utilities are written in various languages, including Python and C.

1.3 Client-Server architecture

The Workbench server has two main parts:

- The *server* manages computationally intensive tasks and user files, and serves application code to the client. The server includes:
  - *Server tools* - set of low-level computational utilities (for instance, sequence designers, behavioral designers, thermodynamic simulators, etc.). These server tools execute *computational tasks*.
  - *Realtime Services* - a set of components which respond to specific user interactions (for instance, requesting creation or modification of files, execution of a computational task, etc.). Each Realtime service may respond to "normal" requests via HTTP, or to events raised via WebSockets.
    * *File service* - provides access to user files
    * *Task service* - initiates and manages computational tasks
    * *Auth service* - authenticates users
  - *Web server* - the server also serves application code and static resources (such as images and CSS files) to the client.
- The *client* provides a rich, graphical interface with which the user can interact through their web browser. The client includes:
  - *Core services* and *Core components* - a set of shared functionality and interface components for interacting with the server (e.g. for managing files and starting computational tasks)
  - *Client-side applications* - a number of rich interfaces for interacting with specific types of files and for performing computational tasks. Examples of application include: the Nodal system editor, the Web Domain Designer (DD), and the Sequence editor.

Additionally, there are several libraries which are shared between the client and the server: - DNA-utils (`dna-utils.js`) - Provides facilities for working with DNA sequences and systems (e.g. complementation, parsing, etc.) - DyNAMiC (`dynamic.js`) - Provides access to the data structures and algorithms of the nodal behavioral compiler - DD (`dd.js`) - The Javascript version of Dave Zhang's Domain Design package

1.4 Javascript Basics

A very cursory overview of the Javascript language is provided in this sections. The reader completely unfamiliar with Javascript is referred to the Mozilla Developer Network's Javascript Guide, which provides a thorough tutorial for beginners. Javascript Garden provides "documentation about the most quirky parts of the JavaScript programming language," including prototypal inheritance, function scope and closures, array iteration, and types/casting.

Javascript is a dynamically-typed, mixed-paradigm language with imperative, object-oriented, and functional aspects. In modern engines, Javascript code

is JIT-compiled, rather than interpreted. Memory is automatically garbage-collected.

1.4.1 Syntax

Javascript syntax is in many ways similar to C; some key differences:

- Variables are declared with the `var` keyword instead of a type (e.g. `var foo;` instead of `int foo;`); no keyword is used before function parameters
- Functions are declared using the `function` keyword, rather than a type; for example: `function foo(bar, baz) { ... }`
- Anonymous functions may also be declared using the function keywork, for instance `function() { ... }`
- Arrays can be declared using square brackets, e.g. `var list = [1, 2, 3, ...];`. Array elements can be accessed as in C (`list[0] = 1;`).
- "Objects," which are really key-value mappings resembling hash tables, can be defined using curly braces, e.g. `var obj = {a: 1, b:2, ...};`. Values of objects can be accessed either using a dot notation like C `structs` (e.g. `obj.a = 1;`) or using an array-like notation (e.g. `obj['a'] = 1;`—note the quotes enclosing the attribute name).
- Single and double-quotes can be used interchangeably, but must be paired.

1.4.2 Functional programming

- Functions are first-class objects in Javascript and may be assigned to variables.
- When functions are defined, a *closure* is created which allows variables in the scope enclosing the function definition to be accessed from within the function.
- Javascript arrays in modern engines contain several functional programming tools, such as `map` and `reduce`, which can be used as follows: `[1, 2, 3].map(function(x) { ... })`, or `[1, 2, 3].reduce(function(total,x) {...},0);`.

1.4.3 Prototypal inheritance

- Javascript classes are just `function` which can be called with the `new` operator. For instance, a `Person` class could be defined as follows: `function Person(name) { this.name = name; }`, and a Person object could be instantiated as `var p = new Person("John Smith");`.
- Javascript classes each have a `prototype` which includes methods and properties shared among all instances of the class. For instance, to give all instances of the `Person` class a `sayName` method, we could use `Person.prototype.sayName = function() { alert(this.name); }`.

- The behavior of the `this` keyword in Javascript is unintuitive to most—`this` is bound to a particular value, for a given function, *at the time the function is called*; this is as opposed to most object-oriented langauges such as C++ where `this` is defined when an object is instantiated. For example, if we were to call `p.sayName()` in a browser after executing the above code, a dialog box would appear containing the text "John Smith". If however we were to execute `var f = p.sayName; f();`, the dialog would instead contain "null". This is because `this` for the invocation `f()` is set to the global object, `window`. See examples on the Javascript Garden for details and further explanation.
- A traditional, class-based inheritance system can be built on top of Javascript's prototypal inheritance system. The Workbench client takes this approach, using Sencha ExtJS's class system to structure code.

2 Server

2.1 Relevant toolkits

As discussed, the server code is executed by the Node.js Javascript runtime. Node.js code is organized into modules according to the CommonJS specification. The Node Package Manager, NPM, provides a repository of modules for Node.js applications. Workbench depends on several NPM modules:

- Connect - a middleware library for error handling, static file serving, etc.
- Express - a web application framework which provides several features: routing HTTP requests to application code based on the request URL; storage of user "session" data; HTTP response handling, etc.
- Socket.io - a WebSockets framework for real-time communication with the server
- Jade - a templating language
- Mongoose - an Object modeling library for the MongoDB NoSQL database.

2.2 Bootstrapping process

This section describes the process by which server code is loaded and executed.

- `workbench.conf` - A script for the event-based task daemon Upstart. The Upstart daemon reads this script which directs it to start the `startup` shell script when the machine boots, and to restart the script when it crashes.
- `startup` - A shell script which exists largely for convenience. The upstart script requires superuser privileges to edit, but this script can live in the

main `webserver-user`'s home directory and therefore be modified without special privileges. This script launches the MongoDB database server which tracks user authentication.

- `app.js` - Loads `connect`, `express`, and a number of other modules, and sets up an HTTP server. Serves a few routes, including the main application page. Loads the following files:
  - `config.js` - Contains configuration variables.
  - `utils.js` - Contains a number of utility functions.
  - `realtime.js` - Contains the Realtime class, which provides an API to configure the Realtime Services. Each of these services is represented as a class which is loaded by `app.js` and passed to the global instance of the Realtime class. Each of the services registers handlers for HTTP requests to some number of URLs, and may register handlers for incoming socket.io events.
    * `task-service.js` - Contains the service which executes computational tasks. The TaskService class communicates with the Realtime class and associates users with the particular tasks that those users(s) are running.
      · `dispatcher.js` - Contains the class responsible for actually *running* the tasks; the Dispatcher class has no knowledge of particular users; it depends on a simple API for connecting the streams exposed by particular task instances to downstream outputs (which, in the case of the Dispatcher instance that runs as part of the TaskService, are socket.io connections to clients.)
    * `file-service.js` - Contains the service which serves user files in response to HTTP requests. Also responds to HTTP requests instructing creation, renaming/moving, updating, and deletion of files.
    * `auth.js` - Contains the service which responds to user authentication requests.

2.3 Realtime services

2.3.1 Task service

- Task classes
  - LocalTask
  - BashTask
  - NodeTask
- Starting tasks
  - Parsing user input data

  - Creating Task object
  - Starting task
  - Associating task with user
  - Piping data to user
  - Watching for task completion
- Stopping tasks

2.3.2 File service

The file service registers handlers for HTTP requests for the following actions:

- Creating files
- Listing contents of a directory
- Reading file contents
- Renaming and moving files
- Updating file contents
- Deleting files

2.4 Server tools

The following computational tools are included:

- Nodal compiler
- Domain-level enumerator
- Pepper compiler
- DD
- Multisubjective
- NUPACK thermodynamics

3 Client

3.1 Relevant toolkits

Client-side code relies heavily upon the following frameworks:

- Sencha ExtJS - a user interface library providing many interface widgets (e.g. buttons, toolbars, windows, layouts, etc.), as well as the class system which structures most of the client-side code
- jQuery - provides HTML DOM traversal and manipulation, event handling, and AJAX communication with the server

- Underscore.js - a functional "utility belt" providing a host of utility functions, including cross-browser implementations of `map` and `reduce`
- CodeMirror - a syntax-highlighting, in-browser code editor.
- D3.js - a library for creating interactive, data-drived visualizations
- Raphael - a library for creating vector graphics in the browser using SVG or VML.

3.2 Lifecycle

- `bootstrap.js`

3.3 Core services

3.4 Client-side applications