DyNAMiC Workbench Architecture

Casey Grun

Contents

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. The server uses a MongoDB non-relational database for storing user login information. A variant of Markdown is used for documentation; Pandoc is used to parse and generate this documentation, as well as user-facing Help documents. Documentation within the code is parsed by JSDuck.

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 section. 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.

1.5 Deployment overview

Generally the Workbench code is deployed on a virtual machine running Ubuntu Server. During development, this is a VirtualBox virtual machine running on the developer's local machine. Ubuntu Server does not come with a graphical shell installed, so the server is generally accessed via SSH or HTTP. Workbench application code is generally placed in a VirtualBox "shared folder" that is synchronized with a folder on the host operating system by VirtualBox; that way, the developer can use his choice of editor in his host operating system, rather than having to edit Workbench code via an editor running over SSH.

The deployment server runs on a virtual machine hosted by Amazon Web Services in the Elastic Compute Cloud ("EC2"). The basic configuration is similar, but there are no shared folders. The EC2 instance uses three Amazon Elastic Block Storage (EBS) volumes for storage: one contains the root file system, one contains user files, and one contains logs. Code is deployed to the server via `git`.

On either the development or the deployment server, code is contained in a subdirectory of the home folder of a non-superuser called `webserver-user`. Sometimes this code may reside elsewhere on the file system (for instance, in the development server it resides in a subdirectory of the mount point for the VirtualBox shared folder), in which case a symbolic link must be created to the appropriate location in the home directory. Likewise, user files are contained in another subdirectory, etc.

The layout of this home directory is as follows (all paths are relative to `/home/webserver-user/`):

- `app/` - Contains application code (layout of this folder is described below)
- `file-share/` - Contains user files within the `files` subdirectory
  - `files/` - Contains subdirectories corresponding to user email addresses. Each of these subdirectories is a Workbench user's "home directory" within the application (e.g. all of that user's files will be stored within this subdirectory; that user will have access only to files within that subdirectory).
- `logs/` - Contains several log files:
  - `startup.log` - Contains the contents of `stdout` from the `startup` shell script (see below)
  - `full.log` - Contains logging information from Workbench application code.
  - `mongo.log` - Contains logging information from the MongoDB database.
- `node_modules/` - Contains node modules installed using the Node Package Manager, `npm`.

1.6 Application Code File Hierarchy

This section outlines the file hierarchy for application code (all paths are relative to `/home/webserver-user/app/`):

- `build/` – Destination for various files generated during the client-side code or development VM build process.
- `help/` - Contains Markdown source for user-facing help documentation files
  - `html/` – Contains HTML code for user-facing help documentation, generated by Pandoc.
- `server/` - Contains server-side application code
  - `node_modules/` - Contains several pieces of shared code which are commonly included in subsequent scripts. Because of the way Node.js resolves module identifiers it is more convenient to have these files in this folder; that way, scripts in subdirectories of `server/` don't have to specify the full relative path to these scripts.
  - `server-tools/` - Contains code for the server-side computational tools.
- `static/` - Contains all static resources that are served to the client
  - `client/` - Contains client-side Javascript code

    - `common/` - Contains Javascript code that is shared between the client and the server. This code is contained below `static/` so that it can be served to the client, but these files are also loaded separately by the server-side code.
    - `docs/` - Contains API documentation generated by JSDuck from DocComment blocks within the application code.
    - `html/` - Contains static HTML pages served to the client
    - `images/` - Contains images and icons
    - `styles/` - Contains CSS stylesheets
    - `tests/` - Contains a small number of unit tests for some common libraries
- `tools/` - Contains executables for the computational tools
- `views/` - Contains template files used to generate the HTML main application page, along with the

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

This section is under construction as this API develops.

- 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

This section is under construction. 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

This section is under construction. 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 File hierarchy

Client-side javascript is organized in the following folder hierarchy (all within the `static/` directory):

- `client/` - Common code, generally under the `App.*` namespace, that is loaded manually and serves to bootstrap the user interface
- `client/ui` - Classes for shared user interface components used by the main application chrome or shared among several applications, generally under the `App.ui.*` namespace.
- `client/usr` - Subdirectories contain client-side applications, under the `App.usr.*` namespace. Each application generally has its own namespace (for instance, the nodal interface resides within `App.sr.nodal.*`.)
- `client/workspace` - Contains code related to the 2D drag-and-drop "workspace" or "canvas" interface presented by applications like the nodal designer. This directory does *not* contain nodal-specific elements, but rather base classes for creating and manipulating vector graphics, equations, etc. Contains classes within the `Workspace.*` namespace.

3.3 Lifecycle

This section describes how client-side Javascript code is loaded and executed. The order in which client-side files are loaded is determined by the *server*-side file `server/manifest.js`. `manifest.js` assembles an array of Javascript files, including all dependencies/libraries, which should be loaded initially into the page. The template `views/index.jade` generates the HTML for the main application page. `index.jade` executes `manifest.js`, and generates `<script>` tags to load the scripts onto the page. Some number of scripts are injected into the page in this way—these are known as "statically-loaded" scripts. *Most* scripts, however, are loaded by `Ext.Loader`, the dynamic dependency-loading mechanism built-in to ExtJS; these are known as "dynamically-loaded" scripts.

During development, this works as follows:

- `loader.js` enables `Ext.Loader`
- Statically-loaded scripts injected via `<script>` tags call methods of `Ext.Loader` to `require` dependency classes. Specifically, `App.Launcher` generally `require`s the root classes corresponding to each of the client-side applications (e.g. `App.usr.nodal.Canvas` represents the Nodal editor, `App.usr.text.Editor` corresponds to a plain text editor, etc.).
- `Ext.Loader` synchonously loads the script files containing these dependencies. The classes described in those scripts may have dependencies, which are resolved by `Ext.Loader` as well.
- This process continues recursively until `Ext.Loader` has resolved all dependencies.

During deployment, the process is a bit different:

- `loader.js` is omitted, so `Ext.Loader` is not enabled
- Prior to deployment, all of the dependency requirements specified within application classes are used to create a graph of dependencies and a proper ordering of all classes contained in dynamically-loaded scripts. This list is combined with the list of statically-loaded scripts in `manifest.js`; all of these files are concatenated to generate one big Javascript file (`static/client/all.js`), which defines all classes needed by the application.
- `all.js` is loaded and executed by the client.

In either case, the following scripts are loaded in approximately this order:

- Statically-loaded scripts:
  - `bootstrap.js` - Loads the ExtJS library
  - Libraries - All other external libraries are loaded, including jQuery, CodeMirror, Raphael, D3.js, etc.
  - `loader.js` - Enables `Ext.Loader`.
  - `app.js` - Contains shared utility functions and code for communicating with the server via Socket.io.
  - `documents.js` - Contains code for managing interactions with the file system. Defines the data model for the Files tree displayed in the main interface.
  - `core.js` - Contains code specifying the data model used within the "workspace" interface appearing in applications such as the nodal UI.
  - `workspace.js` - Contains code initailizing the "workspace" interface
  - `canvas.js` - Contains code which builds the main interface chrome—the files tree, the main tab bar, the console, etc.

    - Common scripts - Scripts from `static/common/` such as `dna-utils.js` and `dynamic.js` are loaded.

- Dynamically-loaded scripts:
    - Shared components - Shared data models and interface widgets from the `client/ui` subdirectory are loaded into the `App.ui.*` namespace.
    - Applications - Client-side application classes are loaded into namespaces underneath `App.usr`.

3.4 Core components and services

The following important classes are used across the application

- `App.Application` - Base class mixed in to all client-side applications. Contains common code for loading and saving data from files. Client-side applications override methods defined on `App.Application` to provide custom behavior, e.g. on file loading
- `App.Document` - The data model which represents an individual file, and communicates changes to that file to the server.
- `App.TaskRunner` - The class which requests execution of computational tasks on the server and manages task updates via Socket.io.
- `App.ui.*` - Shared data models and interface widgets:
    - `App.ui.Launcher` - Manages "launching" of Applications via instantiation of `App.Application` subclasses and association of those instances with `App.Document`s.
    - `App.ui.FilesTree` - Provides a user interface for interacting with user files, including creating, editing, renaming, and deleting files and directories.
    - `App.ui.console.*` - Provides an interactive console to which debug messages are written, and in which arbitrary Javascript code can be executed.

3.5 Client-side applications

This section is under construction.