Introduction

This document contains several sections:

- The **Installation and Source** section explains the structure of InfoMachine's source code and necessary steps to install InfoMachine on a server
- The **Application architecture** section explains how the InfoMachine interface and backend is designed and implemented.

A note on style

`Monospaced` text in this document refers to proper names of objects in code. Class names are capitalized. On the client, method and property names are CamelCase, beginning with lowercase. On the server, method and property names are lowercase, with words separated by underscores.

Compatibility

InfoMachine requires a modern browser with support for scalable vector graphics and CSS3. It has been tested extensively in Google Chrome 8 and Safari

Installation and Source

Credits and dependencies

InfoMachine utilizes the following open source toolkits on the server:

- CodeIgniter (EllisLab) http://codeigniter.com/
- Zend Framework (Zend Technologies) http://framework.zend.com/
- Quickauth (Dave Blencoe) http://www.daveblencowe.com/category/development/codeigniter/quickauth-authentication-library/

On the client:

- jQuery (The jQuery Project) http://jquery.com/
- Ext JS (Sencha) http://www.sencha.com/products/js/

- Rapaël JS (Sencha) http://raphaeljs.com/
- Aloha HTML Editor (Gentics) http://www.aloha-editor.com/
- Mathquill (Jay Adkisson) http://laughinghan.github.com/mathquill/
- Inflection.js (Ryan Schuft) http://code.google.com/p/inflection-js/
- Color.js (Andrew Brehaut) http://www.cs.rit.edu/~ncs/color/t_convert.html
- Date.js (Coolite) http://www.datejs.com/
- String.js (adapted from Prototype.js; Prototype Core Team) http://www.prototypejs.org/

The following artwork:
- Pinvoke fugue icons (Yusuke Kamiyamane) http://p.yusukekamiyamane.com/

File structure

`/canvas` **(root)**
Application root. Contains:
- `index.php` (bootstraps CodeIgniter framework)

/canvas/system
CodeIgniter root. Everything in this folder except the application subdirectory is CodeIgniter framework code; ignore it.

/canvas/system/application
Server-side source code. Contains a lot of files, some of which are configuration, some of which are vestigial, and a few of which are relevant. The relevant ones are:

- `models/workspace_model.php` (workspace model; handles loading and saving of the workspaces. Invoked by the `workspace` controller)

- `views/application_view.php` (main application view; displays interface to logged in user, loads client-side code. Invoked by `application` controller)
- `views/login_view.php` (login view; displays the login and registration windows)

- `controllers/login.php` (performs login functionality; is invoked by code in `login.js`)
- `controllers/application.php` (main application controller—ie: the controller invoked when the user loads the application. Loads the `application_view` view if the user is logged in; otherwise presents the `login_view` view, which allows the user to login or register)
- `controllers/workspace.php` (workspace data controller; loads and saves serialized workspaces)
- `controllers/user.php` (user data controller; passes user data to client-side code)
- `controllers/database.php` (rebuilds the database when invoked)

/canvas/client
Client-side source code (excluding libraries, which are located in `/scripts`). All files in this directory except `login.js` are included by the main `application_view`. Contains:

- `app.js` (application bootstrap and a few utility functions)
- `canvas.js` (contains Ext components which encapsulate workspace interface. Bootstraps the interface by instantiating these components)
- `workspace.js` (contains several core classes, ie: `Workspace` class, which manages the workspace and objects therein; `Workspace.Action` class, which manages changes to the workspace; `Workspace.Components` class which manages serialization and deserialization. Also contains some utility functions in `Workspace.Utils`)
- `objects.js` (contains classes which represent objects in the workspace)
- `tools.js` (contains classes which allow different user interface behaviors within the workspace)
- `login.js` (included by login view; invokes server-side login and account creation handlers)

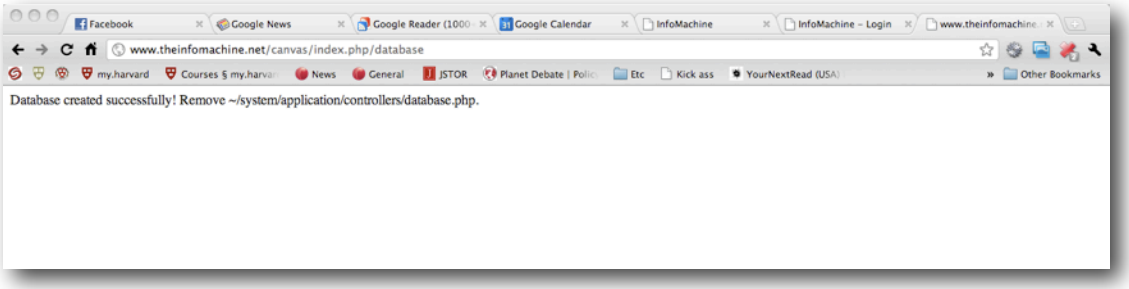
/canvas/styles
Contains CSS styles for the interface

/canvas/icons
Contains images used in the interface

/canvas/etc
Contains extra stuff that isn't actually source code and isn't used by the application.

Installation
The latest version of InfoMachine will be available at http://www.theinfomachine.net/ However, if you must install InfoMachine on another machine or directory, do the following:

- Copy all directories in `~/canvas` to your new directory
- Set appropriate file permissions.
  - The `~/canvas/system` directory needs to be world executable
  - All other directories within ~/canvas need to be world-readable
- Update database connection settings in `~/canvas/system/application/config/database.php`
- Update `base_url` path in `~/canvas/system/application/config/config.php`
- Run the database setup script at `~/canvas/index.php/database`



- Remove the database installation script (or lock it down by setting it to only user-writable)

Application architecture

Key base classes

- `SerializableObject` – base class for Workspace and WorkspaceObject; extends Ext.util.Observable. See below for details.
- `Workspace` – encapsulates the workspace canvas; marshals rendering of HTML and SVG/VML elements to the DOM, handles and dispatches events, and manages creation, serialization, and deserialization of `WorkspaceObjects`
- `WorkspaceObject` – represents a single object in the workspace. Many subclasses implement various types of workspace objects, from sketches, to images, to rich text, to typeset mathematical equations.
- `WorkspaceAction` – represents a change to the workspace. `WorkspaceActions` can be serialized and can automatically generate complementary `WorkspaceActions` which perform undo functions.
- `WorkspaceTool` – represents a set of user interface behaviors within the canvas (call it an “interface mode,” if you will).
- `Ext.ux.LiveCanvas` – exposes the entire workspace interface as an Ext component. Manages the non-canvas UI (the ribbon, the object tree, etc)

User workflow overview

This section is intended to introduce the proper names of some of the relevant classes on the client and server and how they interact in common user interactions. Exhaustive documentation of each class appears in the code, and the details of several key concepts are explained below.

Legend: user action, client action, server action

- `application` controller invoked
  - `login_view` loaded
    - User creates account
    - User logs in
- `application` controller invoked
  - `application_view` loaded
    - Client-side code loaded
      - Serialized workspace data downloaded
        - `workspace` controller’s `load` method is invoked
          - `Workspace_model`’s `load` method is invoked
            - Serialized workspace is loaded from the database
        - Serialized workspace is encoded and delivered to the client in a JSONP file

    - User data downloaded
    - `Ext.ux.LiveCanvas` instantiated
        - Ribbon built
        - `Ext.ux.ObjectTree` built
        - Workspace built
            - `WorkspaceTool`s built
            - Default tool activated
            - WorkspaceObjects deserialized
                - `WorkspaceObject`s instantiated
                - `WorkspaceObject`s `initialize`d
                - `WorkspaceObject`s `render`ed
- User selects a tool
    - `Ext.ux.ToolTab` generates `toolChange` event
        - `Ext.ux.LiveCanvas` activates the new tool using `Workspace.setActiveTool`
            - `Workspace` invokes activate method of the `WorkspaceTool`
- User creates objects
    - `WorkspaceTool` generates `Workspace.Actions.CreateObjectAction`
    - Action is applied to the Workspace
    - Workspace creates objects
        - `WorkspaceObject`s instantiated
        - `WorkspaceObject`s `initialize`d
        - `WorkspaceObject`s `render`ed
    - Workspace interface is updated
- User saves workspace
    - Workspace is serialized
    - `Workspace.objects` is serialized
    - Serialized workspace is posted to server
        - `workspace` controller's `save` method is invoked
            - `Workspace_model`'s `save` method is invoked
                - Serialized workspace is saved to the database

Serialization, deserialization, `wtype`**s**

The user's workspace is saved by "serializing" the `Workspace` object (a complex Javascript object with lots of properties and methods, only several of which really need to be persisted) into a simple Javascript object, containing only the relevant properties and values which must be persisted, and then encoding this object as a JSON string and saving that the server.

`SerializableObject` implements an interface for denoting (using the `expose` method) which properties should be included in the serialized object. It also implements a rudimentary protocol for adding custom getters and setters and fires events when properties are changed.

`SerializableObject.serialize` performs the serialization process for a given `SerializableObject`, by iterating across each of its properties which have been marked as serializable using expose, retrieving them using the accessor, and then serializing those returned values. Values are serialized by testing for presence of the `serialize` method and, if present invoking it (such that `SerializableObjects` contained in the properties of other `SerializableObjects` are also automatically serialized). Values which do not have a serialize method, but are not primitive types like `String`, `Number`, and `Boolean` (that is, regular Javascript `Arrays` and `Objects`) are passed to `Workspace.Components.serialize`, which performs a "deep serialization," searching for any properties of these objects or their descendents which contain a serialize method and applying it in place where necessary.

The object returned from this entire process is known as a "serialized object," and is synonymous with an Ext "configuration object." The `Workspace` itself, and all `WorkspaceObject` classes extend `SerializableObject`. `WorkspaceActions` also extend `SerializableObject`, allowing changes to the Workspace to be serialized and recorded.

To facilitate deserialization, each subclass of `SerializableObject` must registered a string representing its canonical name with `Workspace.Components`. This is string is known as the class's `wtype`, and is generally just the name of the constructor. `wtypes` allow serialized objects to be automatically associated with the appropriate constructors during deserialization. All `SerializableObject` subclasses must then accept a serialized version of that object as the only parameter to their constructor and be able to reconstruct the entire object as it appeared pre-serialization. `SerializableObject` subclasses may do this automatically by invoking `SerializableObject.decode` (which iterates across serialized properties and invokes the setter on the new object instance) in their constructor. However, because `SerializableObject` descendents participate in the `WorkspaceObject` lifecycle (see below), this approach is generally not used, and classes instead implement their own deserialization procedures.

Note: "serialization" may be a bit of a misnomer, since (as noted in the API documentation), the serialize and deserialize methods still deal with Javascript objects, as opposed to strings containing object literals. The process of converting between Javascript objects and strings is referred to as "encoding" for the purposes of this application, and is handled by Ext through Ext.encode and Ext.decode.

WorkspaceObject lifecycle

Constructing a `WorkspaceObject` is a complicated process; `WorkspaceObjects` often contain many properties, as well as some references to other `SerializableObjects` (for instance, `WorkspaceObjects` may have a parent object whose movement they follow; `ConnectorObjects` contain references to the objects they connect, etc.), and certain steps of the process of constructing a `WorkspaceObject` may require all of these properties to be populated in order to function properly. For this reason, requiring, as for a standard `SerializableObject` that deserialization and object set-up happen entirely in

the constructor is inadequate. Rather, all `WorkspaceObject`s implement each of the three methods, which are invoked automatically by `Workspace` in the `createObject` and `createObjects` methods (see API documentation for details):

- (`constructor`) – The constructor should accept a configuration object and save relevant properties to the `WorkspaceObject` instance. The constructor should also expose relevant properties using `expose` and relevant events using `addEvents`
- `initialize` – Resolves any references to other `WorkspaceObject`s. Applied after each object being added to the workspace has been instantiated.
- `render` – Renders a graphical representation of the object to the workspace; `ElementObject`s render an HTML element, while `VectorObject`s generate an Raphaël SVG/VML element. Applied after each object being added to the workspace has been `initialized`.

Note: for the purposes of this application, instantiate refers to simply invoking the constructor, while the terms “build” and “construct” refer to applying all methods of the object lifecycle, in the proper order.

Child objects
Certain `WorkspaceObject` descendents, for instance `WorkspaceIdeaObject`, are designed to contain child objects. While the base `WorkspaceObject` implements functionality for moving child objects along with their parents, subclasses must implement their own methods for adding and deserializing child objects.

The backend
The InfoMachine server component is very dumb, at present; it is essentially responsible for keeping a store of serialized workspaces keyed to user IDs and performing some very basic validation. The `Workspace_model` and `Workspace` controller contain the only real data access code.