The Nature Conservancy

Extending Coastal Resilience

Overview

This document describes the technical design approach Azavea plans to use in developing the Coastal Resilience framework application. The principle design elements are:

- An abstract, configurable site application, which can be skinned and loaded for a distinct geography in the Nature Conservancy's network. Site instances will be separate instances of the same code-base, and operate in isolation of one another so that a user can focus on a particular area of interest.
- A modular, configurable plug-in architecture, which allows specific regional site instances to have tools designed specifically for geo-processing and display. Plug-ins conform to a standard interface which allows them to be reused on other instances, provided that the plug-in developer abstracted relevant data source configuration. Azavea will not develop these major plug-ins, but several of the components of the base framework will be implemented as a plug-ins to serve as reference implementations of the plug-in spec.
- Application resources are served as statically as possible, requiring very low resource requirements. Resources can then be focused on the more intensive geo-processing tasks and map servers.
- A separate, unique application will be developed in conjunction with the regional framework site. This will be a small gateway that can direct a user to a particular regional site, while showcasing the extent of the network. This avoids confusing the "network" concept with a "global" regional site.

Network Site

The network site is a standalone single-page web application that acts as a graphical directory for all the individual map application installations.

Software Stack

The network site is built with HTML, CSS, image files, and JavaScript and does not require any server-side code. To be consistent with the full Coastal Resilience sites, the network site will be served by IIS running on Windows Server.

Deployment

The network site is delivered as an executable Windows installer to allow for simple installation and upgrades. The network site has near-zero resource requirements so it can be deployed to any application server without impacting the performance of other services.

Configuration

The root of the network site application directory contains a sites.json file. This JSON formatted text file (http://www.json.org/) is editable by the site administrator and is used by application to dynamically generate the content of the page.

The sites.json file contains two lists, a list of global sites and a list of regional sites. Each site listed in the sites.json file has the following attributes.

- **`name`** (required)
  The name of the site.
- **description** (required)
  A short paragraph describing the data available on the site.
- **url** (required)
  The full URL of the site
- **bbox** (required for regional sites)
  For regional sites only, an array of latitudes and longitudes (SRID 4326) representing the maximum extent of the site's data.

Example sites.json file

```
{
  "globalSites": [
    {
      "name": "Global",
      "description": "View Global Datasets",
      "url": "http://global.coastalresilience.org"
    }
  ],
  "regionalSites": [
    {
      "name": "Gulf of Mexico",
      "description": "Support Restoration Decisions in the Gulf of Mexico",
      "bbox": [-98.61328125, 17.392579271057766, -79.716796875, 31.653381399664],
      "url": "http://gulfmex.coastalresilience.org"
    },
    {
      "name": "Puget Sound",
      "description": "Support Restoration Decisions in the Puget Sound",
      "bbox": [-123.2720947265625, 47.025206001585396, -121.79443359375,48.111099041065366],
      "url": "http://pugetsound.coastalresilience.org"
    }
  ]
}
```

Regional Site

A regional site is a small ASP.NET application serving a JavaScript client which is configured, skinned and loaded with plugins to operate on a regional geographic extent. A global-scale site is simply a regional site where the "region" is the entire world. All regional sites are created from the same codebase and "skinned" for each region by adding custom configuration files, graphic assets and plug-ins. The plug-ins are similarly configured to operate on the specific regional data relevant to the site into which they are deployed. Since the application processing is almost entirely client-side, the application server will have very low resource requirements. The largest resources should be reserved for the geo-processing tasks on the ArcGIS servers.

Software Stack

Server Application

The site is an ASP.NET MVC application served from IIS running on a Windows Server with the Microsoft .NET Framework, version 4.0 installed. The site is comprised of template pages, configuration files, and plug-ins, which are described in detail in the Plug-in section. The main site page, once constructed, will remain completely static until modifications are made to the configuration, allowing for an aggressive caching strategy, resulting in a highly scalable design.

Database

Since a Regional Site does not do any processing, authentication, session management or other persistence; we don't plan on making use of any traditional relational databases. Plug-ins may utilize their own data sources, however it is assumed that plug-ins will largely interact with ArcGIS Service resources using JSONP.

There may be a simple key-value store database, such as redis, used to store permalink hashes, but the usage is anticipated to be light.

ArcGIS Server

The Regional Site installation will not have any dependencies on ArcGIS Server, assuming that the default basemaps will be served from ArcGIS Online. Plug-ins will contain their own configuration to interact with AGS resources. We recommend ArcGIS Server not be installed on the application server, but on higher resourced machines where they will communicate directly to the client plug-in instances.

Client libraries

Plug-ins are not restricted to using only the JavaScript libraries that the framework app will use, they are allowed to load and use any JavaScript libraries they need to create their UI and process their tasks. The mechanism for doing so is described in the Plug-in section. The framework app will, however, already depend on several libraries and plug-ins are encourage to make use of them to reduce the amount of external code loaded at runtime. They will have access to ArcGIS JavaScript API 3.2, Dojo 1.7.3, and various utility libraries, such as backbone.js and underscore.js.

Configuration

A regional site installation will be configured to represent a specific network site through a JSON file placed in the root of the application directory. This file contains configuration unrelated to the actual application installation details, but rather defines the settings which make the regional sites unique. The elements of the configuration file are:

- **titleMain** (required)
  Name of the regional network, displayed on the application header. May be a link.
- **titleDetail** (required)
  Name of the regional site, displayed on the application header. May be a link.
- **headerLinks**
  An array of dictionaries specifying Text/URL pairs of links to be displayed on the site header. (Optionally the URL may displayed in a popop panel.)
- **initialExtent**
  An array of latitudes and longitudes (SRID 4326) representing the initial extent of the site's map display.
- **basemaps**
  An array of dictionaries specifying Name/URL of ArcGIS Online basemaps
- **pluginOrder** (optional)
  An array of plug-in names in the order that they should appear in the toolbar. If any are specified they will be put in that order, and any other plug-ins loaded will follow in alphabetical order.
- **printServerUrl**
  URL specifying an ArcGIS Server to use for generating PDFs from map layers.
- **googleAnalyticsPropertyId**
  To enable Google Analytics tracking for a regional site: register the site with Google Analytics, record the generated property ID, and use that ID as the value of this property.

An example region.json file:


```
{
  "titleMain": { "text": "Coastal Resilience", "url": "http://coastalresilience.org/" },
  "titleDetail": { "text": "Gulf of Mexico", "url": "http://coastalresilience.org/geographies/gulf-mexico" },
  "headerLinks": [
    { "text": "The Nature Conservency",  "url": "http://www.nature.org/"    },
    { "text": "Partners", "url": "partners.html", "popup": true },
    { "text": "Coastal Resilience",   "url": "http://coastalresilience.org/"   }
 ],
 "initialExtent": [ -98.61328125, 17.392579271057766, -79.716796875,31.653381399664],
 "basemaps": {
  [
    "name": "Topological",
    "url": "http://server.arcgisonline.com/ArcGIS/rest/services/World_Topo_Map/MapServer"
  ],
  [
    "name": "Ocean",
    "url": "http://server.arcgisonline.com/ArcGIS/rest/services/Ocean_Basemap/MapServer"
  ]
 },
 "pluginOrder": [ "layer_selector", "measure", "nearshore_waves" ],
 "printServerUrl": "http://sampleserver6.arcgisonline.com/arcgis/rest/services/Utilities/
PrintingTools/GPServer/Export%20Web%20Map%20Task"
}
```


Skinning

The site will come with a default style specified in custom css files, but the application will include a skin.css file in the client, which can be used to define custom styles for a specific site. The administrator can override existing styles, define new styles and load additional images. All style image resources should be references as URLs relative to skin.css.

Plug-in UIs will inherit styles from the application, but are also encouraged to specify their own specific styling resources in the plug-in folder, as described in the plug-in specification below. Common elements will have styling defined to encourage a consistent style among all tools, but the plug-in is ultimately responsible for constructing its own interface.

Deployment

Each regional or global site is delivered as a distinct, executable Windows installer to allow for simple installation and upgrades. Because all geo-processing is accessed from the browser via JavaScript, the regional sites have near-zero server resource requirements. The runtime performance of the site will not be boosted by being installed on the same application server providing ArcGIS map services and geo-processing for that site. The recommended configuration is to install all regional sites on a small application server configured to scale dynamically in reaction to spikes in traffic.

Framework Application Features and Tools

Permalinks

Application state, including extent, active plugins, selected layers and basemap will be preserved through a URL hashing mechanism, to allow for sharing results and scenarios as a permalink. As described in the plugin specification, when the user requests that a permalink be created for the current application state, the application will poll all loaded plugins via the `.getState()` method. The plugins can provide a data structure of its own design to encapsulate the state of the plugin settings at that time, and can expect to receive that same structure back when a plugin is created and passed the optional state object. The application will handle gathering the state of non-plugin settings and aggregate the information in a URL accessible mechanism that can be shared among users, or bookmarked to revisit a commonly used application state. The state may be encoded and stored entirely as a URL hash, or more succinctly as reference to a stored state on the server.

Logging

Plug-in developers will have access to an error logging service provided through the `.error()`, `warning()` and `info()` methods on the app object passed in on creation. The method will access an endpoint which writes to a log file on the application server for each plugin. A plugin developer can then request access to their plugin's log file to view errors generated on user browsers, greatly enhancing their debugging abilities. The same methods will be used to display consistent info and error message to the user.

Identify

A feature Identify tool will be developed as part of the framework application and will retain styling principles and patterns of the framework components. Identify will work on any map layer provided through an AGS MapServer endpoint which supports the Identify service and is added to the map via the Map Layers tool or a plugin. Administrators can specify field filters, aliases and order through the MXD from which the service was configured, and the Identify results will respect those settings.

Legend

The framework application will provide a Map Legend component that will display legend data for any map layer with legend data that is loaded onto the map.

Split View

The application will provide 2 maps, able to be displayed simultaneously and optionally synchronized to each other. When both maps are displayed, the entire toolbar column and associated tools, including Map Layers and Legend, are duplicated for each map. This structure will allow us to create multiple instances of the same tool, to be able to run simultaneously and have the results visibly compared across maps. Since the active tool's UI is visibly associated with the tool bar and the tool bar with the map, it will be easy for the user to associate an instance of a tool, its settings and its output.

Plug-ins will not need to accommodate the multiple maps design concept, as the framework app will be responsible for creating a plug-in instance with a reference to a map to interact with.

Map Interaction

The framework application will provide basic map tools, including panning, zooming and basemap selection.

Location Search

Each site will be configured with a URL for an ArcGIS Online or locally configured geocoder resource. It will support locating global placenames, disambiguate results and zoom to locations.

Printing/Export

A framework tool will be included which allows the user to draw a polygon over an area of the map and select to export the data layers contained inside, along with some basic template information (Title, Description). ArcGIS 10.1 supports a printing /export service that allows for compositing multiple layer types to various export formats. Due to the disparate sources that plug-ins may use to load layers into the browser's map, there may be some limitations to the exact content that is printable, but it will be supported to the extent that the services are enabled and configured. Plugins that create custom raster images that ArcGIS 10.1 does not include support for exporting would still be free to implement its own custom printing solution.

Map Layers Control

Azavea will develop a plugin that is intended to be used on all regional sites to select and display data layers for a particular region. The plugin itself will contain a "layers.json" configuration file so that it may be redeployed on multiple sites to display local data. The configuration will take an array of AGS MapServer and WMS endpoints and will populate a hierarchical UI to toggle display of all layers contained in the respective endpoint. The URLs must be publicly available, but there are no other restrictions on where they are hosted. The MapLayers plug-in will also offer a text filter to display data layers whose text match the user input.

The configuration file allows the user to specify an AGS or WMS layer source with a required "url" field. Within these sources, an optional "folderTitle" can be included to specify the display name of the folder. Depending on the source type, a "folders" or "layerIDs" field is provided for specifying which services and layers should be included in the UI window, and in what order.

Here is an example layers.json configuration file:

```
[
  {
    "agsSource":
    {
      "url": "http://gulfmex.coastalresilience.org/arcgis/rest/services",
      "folderTitle": "Ags",
      "folders": [
        {
          "name": "Florida"
        },
        {
          "name": "Alabama",
          "services": ["Habitats", "Bathymetry"]
        }
      ]
    }
  },
  {
    "wmsSource":
    {
      "url": "http://preview.grid.unep.ch:8080/geoserver/ows",
      "folderTitle": "Risks",
      "layerIds": [
        "preview:cy_risk",
        "preview:cy_buffers",
        "preview:cs_frequency"
      ]
    }
  }
]
```

Plug-in Specification

The majority of the functionality for a Coastal Resilience regional site (referred to in the rest of this section as "site") is provided by plug-ins.

Conceptually, a plug-in is a bundle of JavaScript code and companion resources which displays a custom HTML-based UI for collecting data from the user, optionally makes requests to geo-processing services, and displays results by creating layers and adding those layers to a common, full-page map.

Physically, a plug-in is a folder containing, at a minimum, a main.js file and an icon.png file. The name of the folder is not significant, but must contain only letters and numbers, no whitespace or punctuation, and should uniquely and succinctly describe the plug-in.

A plug-in is loaded into a site by placing the plug-in folder into the 'plugins' folder in the root application directory and then restarting the application.

main.js
The main.js file is the core of the plug-in and should contain a single AMD-format Dojo module with superclass PluginBase. (http://dojotoolkit.org/documentation/tutorials/1.7/modules)
The module's callback function must return a constructor created by calling `dojo.declare`.

icon.png, icon_active.png
A site shows each available plug-in in a sidebar with an icon and a short name label. The icon must be a 256x256 pixel transparent PNG and should use white only to match the theme of the application. Icon_active.png is shown when the plugin is focused, and icon.png is shown when the plugin is not focused.

plugin.json
This configuration file allows specifying CSS files to load, and specifying module identifiers for non-AMD-compliant JavaScript libraries. For example, here is the plugin.json file for the “layer_selector” plug-in:

```
{
    "css": [
        "plugins/layer_selector/main.css",
        "//cdn.sencha.io/ext-4.1.1-gpl/resources/css/ext-all.css"
    ],
    "use": {
        "underscore": { "attach": "_" },
        "tv4": { "attach": "tv4" },
        "extjs": { "attach": "Ext" }
    }
}
```

The “css” section specifies two CSS files to load – main.css from the layer_selector plugin folder, and ext-all.css from the Sencha CDN repository for the Ext JS library.

The “use” section relates to loading of JavaScript libraries. Plug-in JavaScript files should list libraries in a Dojo “define” statement, but not all third-party libraries support the necessary “Asynchronous Module Definition” (AMD) interface. Listing such a library in the “use” section allows it to be used anyway. For example, the first line in the “use” section above declares that when the library “underscore.js” is referenced as “use!underscore”, its object should be bound to the identifier “_”. For an example of referencing such a library, see the section on “Using Dojo Modules” later in this document.

Additional Files

All resources in the plug-in folder are served publicly. A plug-in can, for example, use images in the plug-in folder to build UI components or require additional Dojo modules stored in separate .js files.

Plug-in Environment

The site loads the ESRI JavaScript API version 3.2, which included Dojo version 1.7.3. Plug-ins should not assume that any other libraries are available and should load any additional requirements as Dojo modules.

Plug-in API

The constructor returned by the Dojo module defined in main.js should have the following static properties:

- **`toolbarName`** (string, required)
  The short string label shown beneath the plug-in icon on the map sidebar
- **`fullName`** (string, required)
  The full description of the plugin, used as a tooltip
- **`toolbarType`** (string, required)
  Either “sidebar” to show the plugin icon in the sidebar, or “map” to overlay the plugin icon on the top area of the map.
- **`showServiceLayersInLegend`** (boolean)
  true to show Legend info for layers added by the plugin, or **false** to omit. Default **true**.
- **`allowIdentifyWhenActive`** (boolean)
  true if clicking on the map should perform an “identify” operation when the plugin is active, or **false** if not. Default **false**.

These properties are inspected by the application and used to build the UI for activating the plug-in.

When a user selects a plug-in to be added to the map, the application constructs a new instance of the plug-in and passes the following options through to the `initialize()` function:

- **`app`** - always provided
  An object that provides access to common application functionality.
  - **`version`**
    Property that returns the version string for of the host site
  - **regionData**
    An object containing the application configuration from the site's region.json
  - **`info`**`(userMessage, developerMessage)`
    `warning``(userMessage, developerMessage)`
    `error``(userMessage, developerMessage)`
    Functions used to display and/or log messages using the common alert and logging functionality provided by the site. A non-empty string passed as the `userMessage` argument will be shown to the user as a pop-up message. A non-empty string passed as the `developerMessage` argument will be written to a server-side log file. For example, developer messages for a plug-in named "zoom_to" are written to the file "zoom_to.log" in the "logs" folder of the main web application directory.
  - **`_unsafeMap`**
    Provides access to the underlying ESRI JavaScript API map object for the application. Do not use this property if it can be avoided.
  - forceDeactivate()
    Call this method to tell the framework that your plugin is no longer active
- **`map`** - always provided
  An object that allows the plug-in to interact with the map and respond to map events. Exposes most of the properties and methods of the ESRI JavaScript map API (http://help.arcgis.com/en/webapi/javascript/arcgis/help/jsapi_start.htm#jsapi/map.htm)
  Methods related removing or modifying layers are restricted to the layers that have been added by the plug-in.
- **`container`** – always provided
  The DOM element in which the HTML UI for interacting with the plug-in should be placed
- **`legendContainer`** – always provided
  A DOM element into which the plug-in may put HTML to be shown in the map Legend.

If any of the following instance methods are defined by a the plug-in instance they will be called as the user interacts with the site.

- **`activate`**`()`
  Called when the plug-in is focused as a results of the user clicking the plug-in icon in the application sidebar. In this method the plug-in may render user interface controls in its provided container div, connect map and mouse events, and add layers to the map.
- **`deactivate`**`()`
  Called when the plug-in loses focus, most often when another plug-in is selected. In this method the plug-in should disconnect any events. Generally the plug-in should not hide map layers here as they may be important to see while using other plug-ins.
- **`getState`**`()`
  Called when the user wants to save the state of the application as a permalink. Should return an object that can be used to restore all settings.

- **`hibernate`**`()`
  Called when the user explicitly "turns off" the plug-in. In this method the plug-in should remove (or hide) amy map layers it has added.
- **`identify(point, processResults)`**
  Called when the user clicks on the map. In this method the plug-in should identify features at the given **point** and pass an HTML summary to the **`processResults`** function, which takes the following arguments:
  - **results** (required) – HTML DOM elements summarizing information at the given point, or **false** if the plug-in has no elements at the point.
  - **width** (optional) – Desired width of the "identify" popup.
  - **height** (optional) – Desired height of the "identify" popup.

  If this function is omitted, default information will be shown for any layers added by the plug-in.
- **`setState`**`(state)`
  Called after **`initialize`** if the application was launched from a permalink. `state` contains the object returned by the plug-in's **`getState`** method when the permalink was created.

Plug-in Responsibilities

A plug-in is responsible for:

- Generating the HTML UI to adjust all plug-in settings and adding the UI to the container div provided.
- Creating layers for displaying data and adding those layers to the map.
- Restoring all settings and layers from a saved state object.
- Logging service failure details, informational messages, and unexpected warnings or errors by calling the common logging methods provided by the `app` object passed to the plug-in constructor.
- Loading additional library dependencies from the plug-in directory as Dojo modules.

A plug-in is *not* responsible for:

- Identifying features on the map and displaying returned attributes.
- Adding base maps.
- Showing a legend.

Plug-In Workflow Example

1. A user visits the site.
2. The site enumerates the sub- folders in the plugins folder and, for each one, dynamically generates JavaScript to load the module defined in main.js and uses the static properties of the loaded module to display the plug-in names on the sidebar.
3. User clicks a plug-in
4. The site creates an empty DOM element in a pop-up panel then constructs a new instance of the plug-in, passing it the DOM element along with `app` and `map` instances.
5. The plug-in adds UI to the DOM element provided and adds a layer to the map.
6. The user interacts with the UI in the pop-up panel.
7. The plug-in handles the UI events by changing the properties of the layer, triggering a refresh of the map.
8. The user clicks a different plug-in icon in the sidebar.
9. The plug-in's deactivate() method is called.
10. The site hides the pop-up panel for the first plug-in and shows a pop-up panel for the newly selected plug-in.
11. The user clicks the first plug-in icon in the sidebar.
12. The site hides the pop-up panel for the second plug-in and shows the pop-up panel for the original plug-in.
13. The site calls the `activate()` method on the plug-in.
14. The user clicks the "Save/Share" button on the site.
15. The site calls the plug-in's `getState()` method and the plug-in returns an object containing all the values needed to restore the UI settings.
16. The site combines all the state information into a hash and shows the user a url for bookmarking or sharing.

Plug-In Best Practices

Define `getState()` and `setState()` instance methods so that all the settings for the plug-in can be saved and restored.

Include a version number in the state object returned from `getState()`. An updated version of the plug-in with a different state object schema may be constructed with a state object in an older format if a user follows an old permalink bookmark. Having a version number embedded in the state makes it easier to convert the schema of the state object if required.

Do not create any DOM elements outside the container element provided to the plug-in constructor.

Define `activate()`, `deactivate()` and `hibernate()` instance methods which enable and disable event handling as the plug-in gets and loses focus. If the plug-in, for example, runs a calculation and adds point to the map whenever a user clicks the map, make sure to disable the click handler when `deactivate()` is called so that the user is not confused by additional dots when they have switched to using a different plug-in.

It is okay to zoom the map to the extent covered by the plug-in when the plug-in is first constructed, but avoid any automatic map movement after that point. Users are often confused when the map moves unless they directly manipulate the map or select a "zoom to extent" type option.

Keep all references to external service URLs outside of the plug-in code and, instead, define them in a JSON-format configuration file. This makes it easier to update the plug-in if a server changes address and also makes it easy to repackage a plug-in for use in different regions by simply changing the data sources.

The larger a Dojo module is, the more difficult it is to maintain. Try to break a large module into smaller modules and use `dojo.require` to join them together at runtime.

The 'map' object provided to a plug-in instance is not an ESRI JavaScript API map instance, but a proxy that attempts to prevent plug-ins from conflicting with one another or the host application. If your plug-in needs access to a method or event that is not exposed by the proxy 'map' object the `map._unsafeMap` property provides direct access to the underlying ESRI map object. Avoid using this property and, instead, contact the application maintainers and request an update to the map proxy that enables the additional functionality.

Do not embed any sensitive information like authentication credentials in any of the files in the plugin folder. All files in the plugin folder are publicly available and can be read by any user.

Make the plugin icon unique, but not too detailed so that users will recognize the tool at a glance.

Use ArcMap to edit the layer properties for any map services used by the plug-in to provide attribute name aliases, reorder attributes, and hide unnecessary or confusing attributes so that the identify results are easy to read and understand.

Plug-in Example: The Measure Tool

By default all sites will include a Measure tool implemented as a plug-in. This provides useful functionality for the application and also serves as a reference implementation for other plug-in developers.

The Measure tool UI will show a toggle button to switch between "length" and "area" modes, a drop-down to select the unit of measurement, and a button to clear any existing measurement and start a new one. The plug-in adds a new Graphics layer to the map and adds geometry to it as the user clicks on the map. Each click on the map updates triggers an update to a label on the UI with the total measurement. If the user selects another plug-in, the Measure plug-in hides the Graphics layer and restores it if the plug-in is re-selected.

Example JavaScript Code: Using Dojo Modules

Plug-in JavaScript should be self-contained, relying on the framework only for loading the ArcGIS JavaScript API and the plug-in's “main.js” file. The plug-in should explicitly reference all other JavaScript files and third-party libraries, even if other plug-ins or the framework itself also references them. This allows plug-ins to be easily portable between sites, and to be insulated from future framework changes.

To accomplish this, plug-ins should rely on the Dojo “require”, “define”, and “declare” statements. Consider this example plug-in “main.js” file:

```
 1  require({
 2      packages: [
 3          {
 4              name: "jquery",
 5              location: "//ajax.googleapis.com/ajax/libs/jquery/1.9.0",
 6              main: "jquery.min"
 7          },
 8          {
 9              name: "underscore",
10              location: "//cdnjs.cloudflare.com/ajax/libs/underscore.js/1.4.4",
11              main: "underscore-min"
12          }
13      ]
14  });
15
16  define(
17      ["dojo/_base/declare",
         "framework/PluginBase",
18        "./ui",
19        "jquery",
20        "use!underscore"
21      ],
22      function (declare, PluginBase, ui, $, _) {
23
24          return declare(PluginBase, {
25              toolbarName: "Map Layers",
26              toolbarType: "sidebar",
27
28              _ui: null,
29
30              initialize: function (frameworkParameters) {
31                  declare.safeMixin(this, frameworkParameters);
32                  this._ui = new Ui(this.container, this.map);
33              },
34
35              activate: function () {
36                  this._ui.display();
37              }
38
39              // ... other code, referencing JQuery using "$"
40              // and referencing underscore.js using "_"
41              // ...
42
43          });
44      }
45  );
```

The “packages” statement (lines 2-13) specifies library module names and locations. Plug-ins should list all referenced libraries here, whether the library is used in this file or in another of this plug-in's JavaScript files. Here both JQuery and Underscore.js are referenced from online CDN's.

The “define” statement 's first argument (lines 17-21) specifies which modules are used in this file. Note that “./ui” refers to the file “ui.js”, another JavaScript file in this plug-in's folder. JQuery is referenced using the package name “jquery” specified on line 4. Underscore.js is likewise referenced using the package name “underscore” specified on line 9, but it is prefixed with “use!”. That's because like many libraries, Underscore.js does not support the “Asynchronous Module Definition” (AMD) interface needed for use with Dojo's “define” statement. Prefixing with “use!” (and listing the library in the “use” section of the plug-in's configuration file “plugin.json”) allows it to be used.

The “function” statement (line 22) contains an argument for each of these modules, binding the module's primary object to the specified JavaScript variable. For example, JQuery is bound to “$”.

Dojo ensures that all 4 modules are loaded. This file's JavaScript code can refer to them using the specified variables.