UIC Architecture

# Introduction

This document proposes an architecture for the next generation of the Tealeaf UI SDK library. The architecture is based on review of the current (as of March 2012) Tealeaf UI SDK library and the next generation architecture designed by Manoj Punjabi.

The architecture design is based partly on Manoj’s architecture design and partly on a reference architecture created and first published in 2009 by Nicholas Zakas in the presentation, Scalable JavaScript Application Architecture (<http://www.slideshare.net/nzakas/scalable-javascript-application-architecture>).

Note: This is a high-level architecture document intended to give general instruction. Implementors may find the need to modify individual component interfaces. The interfaces have intentionally been left sparse so as to allow room for extension.

# Definitions

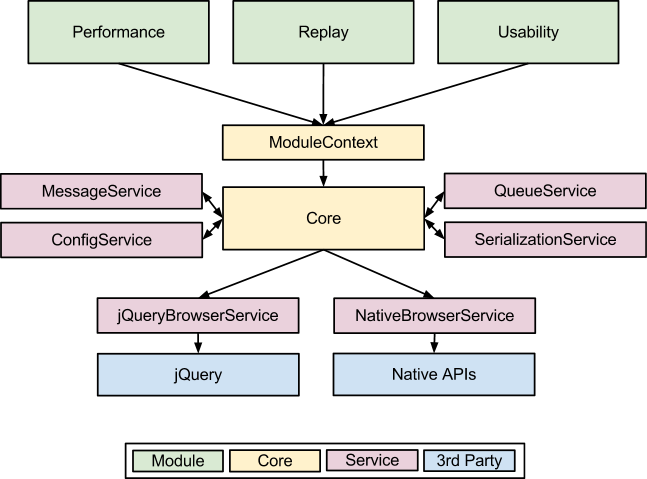
* **Base** - the combination of the Core and required services.
* **Component** - one part of the overall system. The Core, each module, and each service are all considered components.
* **Core** - the object responsible for running the code on a page. It manages module registration and lifecycle as well as service registration. Once written, the Core rarely changes.
* **Module** - an independent unit of functionality in the system whose primary purpose is to monitor and record information about a web page. Modules can never interact directly with other modules. Modules may be allowed to access other types of components indirectly but never directly.
* **Service** - an object necessary for the Core to do its job. Services are able to be swapped in and out so long as they adhere to certain interfaces. This is how the Core is able to extend and change over time. Services may interact with the Core and other services.
* **System** - the combination of all components (Core, modules, and services).

# Goals

1. Each component must be independently testable, even without other components available on the same web page.
2. Each component must be able to be developed separately, potentially by separate teams in separate locations.
3. Modules must not have direct knowledge of one another.
4. Modules must be able to be loaded, enabled, and disabled dynamically during the lifecycle of a web page.
5. Modules must not have direct knowledge of which browser is being used.
6. Once created, this system should not need to be completely rewritten for at least three years. Individual components may, however, be replaced or rewritten.

# High Level Design

# 



The most important part of the system is the Core, which manages everything else in the system. It’s primary job is to manage services and modules, including module lifecycle. In this way, the Core is actually not responsible or aware of business logic, allowing it to be easily extended. Each service acts as a plugin for the Core, providing additional functionality such as a browser normalizing API. Services are utilities designed to be used by the system.

Modules provide the actual business logic for the system by using the module infrastructure and optionally some services to perform their tasks. Modules must not know about one another in order to remain easily pluggable. Modules must also never access the Core directly. This allows the Core, and all services, to be swappable or even removed without requiring changes to modules. Modules must also never access global variables directly, as this makes it more difficult to recreate a suitable environment for unit testing. If a module needs access to a global object, it must request such access through ModuleContext.

This architecture is extensible in two ways: services extend the Core to provide more functionality and modules extend the system by providing more business logic. Doing so makes it easy to add new capabilities to the system without needing to rewrite from scratch. Additionally, this level of abstraction makes it easier to reliably test each component without needing the complete system to be present.

# Interfaces

The interfaces for the UI SDK architecture are defined below in WebIDL (<http://www.w3.org/TR/WebIDL/>) format.

## Core

The Core interface represents the controller of the system. It is responsible for managing configuration, services, and modules.

interface Core : ConfigService {

// Core Lifecycle

void destroy();

boolean isInitialized();

void init(SystemConfig config);

// Module Lifecycle

boolean isStarted(DOMString moduleName);

void start(DOMString moduleName);

void startAll();

void stop(DOMString moduleName);

void stopAll();

// Data Retrieval

Module getModule(DOMString moduleName);

Object getService(DOMString serviceName);

// Messaging

void broadcast(Message message);

void listen(DOMString moduleName, DOMString messageType);

// Registration

void addModule(DOMString moduleName, ModuleCreator create);

void addService(DOMString serviceName, ServiceCreator create);

// Unregistration (primarily for testing)

void removeModule(DOMString moduleName);

void removeService(DOMString serviceName);

// Queues

void post(DOMString moduleName, QueueEvent event,

optional DOMString queueId);

};

### destroy()

Instructs the system to shutdown. Stops all modules and unhooks any events.

**Parameters:**

(none)

**Returns:**

(nothing)

### isInitialized()

**Parameters:**

(none)

**Returns:**

True if the system is initialized, false if not.

### init()

Initializes the system, passing the configuration settings to ConfigService and signals modules to start (calls startAll()).

**Parameters:**

* config - the global configuration information used by the system.

**Returns:**

(nothing)

### isStarted()

Determines if a given module is already started.

**Parameters:**

* moduleName - the name of the module to check.

**Returns:**

True if the module is started, false if not.

### start()

Starts a single module, ensuring its events are hooked up and calls Module::init().

**Parameters:**

* moduleName - the name of the module to start.

**Returns:**

(nothing)

### startAll()

Starts all modules registered in the system, calling Module::init() on each.

**Parameters:**

(none)

**Returns:**

(nothing)

### stop()

Stops a single module, ensuring its events are unhooked and calls Module::destroy().

**Parameters:**

* moduleName - the name of the module to stop.

**Returns:**

(nothing)

### stopAll()

Stops all modules registered in the system, calling Module::destroy() on each.

**Parameters:**

(none)

**Returns:**

(nothing)

### getModule()

Returns the Module object associated with a given name.

**Parameters:**

* moduleName - the name of the module to retrieve.

**Returns:**

The Module object for the given name.

### getService()

Returns a service object identified by the given name.

**Parameters:**

* serviceName - the name of the service to retrieve.

**Returns:**

The service object or null if not found.

### broadcast()

Broadcasts a message throughout the system to whichever components are interested in it.

**Parameters:**

* message - the message to send.

**Returns:**

(nothing)

### listen()

Instructs the Core to direct messages of a certain type to the specific module.

**Parameters:**

* moduleName - the name of the modules that wants to listen for the message.
* messageType - the type of message to listen for.

**Returns:**

(nothing)

### addModule()

Registers a module with the Core but does not start it.

**Parameters:**

* moduleName - the name of the module to register.
* create - the function used to create the module object.

**Returns:**

(nothing)

### addService()

Registers a service with the Core.

**Parameters:**

* serviceName - the name of the service to register.
* create - the function used to create the serviceobject.

**Returns:**

(nothing)

### removeModule()

Stops and ruregisters a module with the TestCore. The module is first stopped and then removed.

**Parameters:**

* moduleName - the name of the module to unregister.

**Returns:**

(nothing)

### removeService()

Unregisters a service with the TestCore.

**Parameters:**

* serviceName - the name of the service to unregister.

**Returns:**

(nothing)

### post()

Send event information to the module’s default queue. This doesn’t necessarily force the event data to be sent to the server, as this behavior is defined by the queue itself. Calls QueueService to handle the functionality.

**Parameters:**

* moduleName - the name of the module saving the event.
* event - the event information to be saved to the queue.
* queueId - (Optional) specifies the ID of the queue to receive the event.

**Returns:**

(none)

## TestCore

The TestCore interface represents an object for testing components. This would never be deployed to a browser, but is used purely for automated testing without loading the entire system.

interface TestCore {

// Data Management

Module getModule(DOMString moduleName, Object context);

Object getService(DOMString serviceName, Object context);

// Registration

void addModule(DOMString moduleName, ModuleCreator create);

void addService(DOMString serviceName, ServiceCreator create);

// Unregistration

void removeModule(DOMString moduleName);

void removeService(DOMString serviceName);

};

### getModule()

Returns the Module object associated with a given name.

**Parameters:**

* moduleName - the name of the module to retrieve.
* context - a placeholder for the ModuleContext object.

**Returns:**

The Module object for the given name.

### getService()

Returns a service object identified by the given name.

**Parameters:**

* serviceName - the name of the service to retrieve.
* context - a placeholder for the ServiceContext object.

**Returns:**

The service object or null if not found.

### addModule()

Registers a module with the TestCore but does not start it.

**Parameters:**

* moduleName - the name of the module to register.
* create - the function used to create the module object.

**Returns:**

(nothing)

### addService()

Registers a service with the TestCore.

**Parameters:**

* serviceName - the name of the service to register.
* create - the function used to create the service object.

**Returns:**

(nothing)

### removeModule()

Unregisters a module with the TestCore. The module is first stopped and then removed.

**Parameters:**

* moduleName - the name of the module to unregister.

**Returns:**

(nothing)

### removeService()

Unregisters a service with the TestCore.

**Parameters:**

* serviceName - the name of the service to unregister.

**Returns:**

(nothing)

## Module

The Module interface represents a single module. It defines how modules interact with the system. Even though modules may have very different functional purposes, they all interact with the system in the same manner. Each of the methods in this interface are not designed to be called by the module itself but rather by the Core as it manages the module’s lifecycle. Modules specify which events they want to listen to in their configuration, not in the JavaScript logic.

interface Module {

// Lifecycle Methods

void destroy();

void init();

// Events and Messaging

void onevent(WebEvent event);

void onmessage(Message message);

};

### destroy()

Destroys the module. Everything the module needs to do upon starting up should be executed in this method. This method is called when it’s time for the module to stop.

**Parameters:**

(none)

**Returns:**

(nothing)

### init()

Initializes the module. Everything the module needs to do upon starting up should be executed in this method. This method is called when it’s time for the module to start.

**Parameters:**

(none)

**Returns:**

(nothing)

### onevent()

This method is called when a browser event occurs that the module wants to track. The events are part of the module’s configuration.

**Parameters:**

* event - an object containing all of the relevant meta data for the browser event.

**Returns:**

(nothing)

### onmessage()

This method is called when a system message occurs that the module wants to track. Modules can listen for specific messages by using their ModuleContext object.

**Parameters:**

* message - an object containing the message information.

**Returns:**

(nothing)

## Message

The Message interface represents a system message. System messages are broadcast to all components to signal important moments.

interface Message {

attribute DOMString type;

attribute any data;

};

### type

A string identifier for the message.

### data

Any data related to the message. The exact data type is determined by the type of message.

## ModuleCreator

The ModuleCreator interface is a function that is used to create module instances.

[NoInterfaceObject]

interface ModuleCreator {

Module create(ModuleContext context);

};

**Parameters:**

* context - the ModuleContext object providing the module’s view into the system.

**Returns:**

The Module object representing the module.

## ServiceCreator

The ServiceCreator interface is a function that is used to create service objects.

[NoInterfaceObject]

interface ServiceCreator {

Object create(ServiceContext context);

};

**Parameters:**

* context - the object providing the service with an interface to the system.

**Returns:**

Any object. Services simply encapsulate functionality to be used by the Core.

## ModuleContext

The ModuleContext object is a facade that gives modules indirect access to the Core. This object already knows the module name, so the module should never have to identify itself.

interface ModuleContext {

// Meta Data

ModuleConfig getConfig();

// Messaging

void broadcast(Message message);

void listen(DOMString name);

// Event Management

void post(QueueEvent event, optional DOMString queueId);

};

### getConfig()

Returns the configuration object for the module.

**Parameters:**

(none)

**Returns:**

The complete configuration object for the module.

### broadcast()

Broadcasts a message throughout the system.

**Parameters:**

* message - the message to be sent. The data property may contain any type of data.

**Returns:**

(none)

### listen()

Informs the system to signal the module when a given message type occurs.

**Parameters:**

* name - the type of message to listen for.

**Returns:**

(none)

### post()

Send event information to the module’s default queue. This doesn’t necessarily force the event data to be sent to the server, as this behavior is defined by the queue itself.

**Parameters:**

* event - the event information to be saved to the queue. The module must create this object.
* queueId - (Optional) specifies the ID of the queue to receive the event.

**Returns:**

(none)

## ServiceContext

The ServiceContext object is a facade that gives services indirect access to the Core.

interface ServiceContext : Core {

// TBD

};

## ModuleConfig

The ModuleConfig interface represents configuration data for a single module.

interface ModuleConfig {

// TBD

};

## GlobalConfig

The GlobalConfig interface represents configuration data for all components in the system.

interface GlobalConfig {

// TBD

};

## WebEvent

The WebEvent interface represents a normalized browser event object. When an event occurs, the BrowserService wraps the native event object in a WebEvent.

interface WebEvent {

attribute boolean custom;

attribute Event nativeEvent;

attribute Point position;

attribute ElementData target;

attribute int timeStamp;

attribute DOMString type;

};

### custom

True if the event is a custom event from the system, false if the event is a browser-generated event.

### nativeEvent

If custom is false, then this contains a reference to the browser-generated event object. If custom is true, this is null.

### position

Contains position information for the event.

### target

Meta information about the target element for the event.

### timeStamp

The time at which the event occurred in milliseconds since January 1, 1970.

### type

The name of the event.

## Point

The Point interface represents a specific point on the page in x- and y-coordinates.

interface Point {

attribute int x;

attribute int y;

};

### x

The x-coordinate of the point.

### y

The y-coordinate of the point.

## ElementData

The ElementData interface represents a normalized browser event object.

interface ElementData {

const unsigned short HTML\_ID = 1;

const unsigned short XPATH\_ID = 2;

const unsigned short ATTRIBUTE\_ID = 3;

attribute HTMLElement element;

attribute DOMString id;

attribute unsigned short idType;

attribute DOMString type;

attribute DOMString subType;

attribute Object state;

attribute Point position;

attribute Size size;

};

### id

A unique identifier for the element. If idType is HTML\_ID, then this is the value of the id HTML attribute; if idType is XPATH\_ID, then this is an XPath for the element; if idType is ATTRIBUTE\_ID, then this is the value of the custom attribute.

### idType

A constant value indicating the type of string identifier stored in id. Either HTML\_ID or XPATH\_ID or ATTRIBUTE\_ID.

### type

The primary category of the element (normally the HTML tag): input, a, img, div etc.

### subType

The subcategory of the element such as radio, checkbox, password etc. Not all elements have a subType.

### state

The value of input controls. For checkboxes and radio buttons, there is also a property indicating if it is checked.

### position

Position of the element on the page. Exact details TBD.

### size

The element’s width and height on the page. Exact details TBD.

## Size

The Size interface represents the width and height of an element on the page.

interface Size {

attribute int height;

attribute int width;

};

### height

Height of the element that received the event.

### width

Width of the element that received the event.

## QueueEvent

The QueueEvent is a minimal interface for events that can be posted to a queue.

interface QueueEvent {

// TBD

};

## QueueService

The QueueService manages all queues in the system.

interface QueueService {

// force flush

void flush (DOMString queueId);

void flushAll();

// add to queue

void post(DOMString moduleName, QueueEvent event,

optional DOMString queueId);

};

### flush()

Forces a particular queue to be flushed, sending its information to the server.

**Parameters:**

* queueId - the ID of the queue to be flushed.

**Returns:**

(nothing)

### flushAll()

Forces all queues to be flushed, sending all queue information to the server.

**Parameters:**

(none)

**Returns:**

(nothing)

### post()

Send event information to the module’s default queue. This doesn’t necessarily force the event data to be sent to the server, as this behavior is defined by the queue itself.

**Parameters:**

* moduleName - the name of the module saving the event.
* event - the event information to be saved to the queue.
* queueId - (Optional) specifies the ID of the queue to receive the event.

**Returns:**

(none)

## MessageService

The MessageService creates messages in the correct format to be transmitted to the server.

interface MessageService {

Object createMessage(QueueEvent event);

};

### createMessage()

Accepts a simple queue event and wraps it into a complete message that the server can understand.

**Parameters:**

* event - the simple event information.

**Returns:**

A complete message that is ready for transmission to the server.

## BrowserService

The BrowserService object is an abstraction of native browser capabilities. This interface may be implemented either by using an existing framework, such as jQuery, or by writing directly to native APIs. There will be one BrowserService per JavaScript library (i.e., jQueryBrowserService, YUIBrowserService, etc.).

interface BrowserService {

// DOM retrieval

HTMLElement query(DOMString query, optional HTMLElement scope,

optional DOMString type);

HTMLElement[] queryAll(DOMString query, optional HTMLElement scope,

optional DOMString type);

// Ajax

void loadScript(DOMString url);

void sendRequest(AjaxRequest message);

// Events

void subscribe(DOMString eventName, Object target,

EventHandler handler);

void unsubscribe(DOMString eventName, Object target,

EventHandler handler);

};

### query()

Finds a single element in the DOM matching a particular query.

**Parameters:**

* query - either a CSS or XPath query.
* parent - (Optional) the DOM subtree to run the query in. If not provided, document is used.
* type - (Optional) the type of query. Either "css" (default) or "xpath" to allow XPath queries.

**Returns:**

The first matching HTML element or null if not found.

### queryAll()

Finds all elements in the DOM matching a particular query.

**Parameters:**

* query - either a CSS or XPath query.
* parent - (Optional) the DOM subtree to run the query in. If not provided, document is used.
* type - (Optional) the type of query. Either "css" (default) or "xpath" to allow XPath queries.

**Returns:**

An array of HTML elements matching the query or an empty array if not found.

### loadScript()

Loads a JavaScript resource onto the current page.

**Parameters:**

* url - the URL of the JavaScript file to load.

**Returns:**

(nothing)

### sendRequest()

Makes an Ajax request to the server.

**Parameters:**

* message - all of the details about the request.

**Returns:**

(nothing)

### subscribe()

Subscribes an event handler to be called when a particular event occurs.

**Parameters:**

* eventName - the name of the event to listen for.
* target - the object on which the event will fire.
* handler - the function to call when the event occurs. The BrowserService passes a WebEvent object to this handler (not the native DOM event).

**Returns:**

(nothing)

### unsubscribe()

Unsubscribes an event handler from a particular event.

**Parameters:**

* eventName - the name of the event for which the handler was subscribed.
* target - the object on which the event fires.
* handler - the function to remove as an event handler.

**Returns:**

(nothing)

## AjaxRequest

The AjaxRequest interface specifies all of the information necessary to make an Ajax request to the server.

dictionary AjaxRequest {

DOMString? contentType;

DOMString? data;

Object? headers;

AjaxResponseHandler oncomplete;

int? timeout;

DOMString? type;

DOMString url;

};

### contentType

Optional. Set to a string to override the default content type of the request.

### data

Optional. A string containing data to POST to the server.

### headers

Optional. An object whose properties represent HTTP headers.

### oncomplete

A callback function to call when the request has completed.

### timeout

The number of milliseconds to wait for a response before closing the Ajax request.

### type

Either “GET” or “POST”, indicating the type of request to make. Default is “POST”.

### url

The URL to send the request to. This should contain any required query string parameters.

## AjaxResponseHandler

The AjaxResponseHandler interface is a function that receives a server response after an Ajax request, handling both the success and failure cases.

callback interface AjaxResponseHandler {

void handleAjaxResponse(AjaxResponse response);

};

**Parameters:**

* response - a normalized response object containing information from the server.

**Returns:**

(nothing)

## AjaxResponse

The AjaxResponse interface contains data sent back from the server in response to an Ajax request.

interface AjaxResponse {

attribute Object headers;

attribute DOMString responseText;

attribute int statusCode;

attribute boolean success;

};

### headers

An object whose properties are HTTP response headers.

### responseText

The raw text returned from the server or null if the server response wasn’t received.

### success

True if the request completed successfully (status code of 200) or false if not (other status codes or errors).

### statusCode

The HTTP status code of the response.

## EventHandler

The EventHandler interface is a function that receives events from the page.

callback interface EventHandler {

void handleEvent(Event event);

};

**Parameters:**

* event - an event object containing information about the event. This may be a WebEvent object or another type of event object.

**Returns:**

(nothing)

## EventTarget

The EventTarget interface is an abstract base upon which others can inherit to provide event information.

interface EventTarget {

void publish(DOMString eventName, optional Object data);

void subscribe(DOMString eventName, EventHandler handler);

void unsubscribe(DOMString eventName, EventHandler handler);

};

### publish()

**Parameters:**

* eventName - The name of the event to publish.
* data - (Optional) An object containing extra data for the event (if required).

**Returns:**

(nothing)

### subscribe()

**Parameters:**

* eventName - the name of the event to subscribe to.
* handler - the function to call when the event occurs.

**Returns:**

(nothing)

### unsubscribe()

**Parameters:**

* eventName - the name of the event to unsubscribe from.
* handler - the function to remove as an event handler.

**Returns:**

(nothing)

## SerializerService

The SerializerService provides the ability to serialize data into one or more string formats.

interface SerializerService {

Object parse(DOMString data, optional DOMString type);

DOMString serialize(Object data, optional DOMString type);

};

### parse()

Parses a string into a JavaScript object.

**Parameters:**

* data - the string to parse.
* type - (Optional) the format of the data. Default is "json".

**Returns:**

An object representing the string data.

### serialize()

Serializes object data into a string using the format specified.

**Parameters:**

* data - the data to serialize.
* type - (Optional) the format to serialize the data into. Default is "json".

**Returns:**

A string containing the serialization of the data.

## ConfigService

The ConfigService is responsible for managing global configuration settings. This may include receiving dynamic configuration updates from the server at regular intervals. The ConfigService fires a configupdated event when it receives updated configuration information.

interface ConfigService : EventTarget {

// Global Configuration

GlobalConfig getConfig();

void updateConfig(GlobalConfig config);

// Core Configuration

CoreConfig getCoreConfig();

void updateCoreConfig(CoreConfig config);

// Module Configuration

ModuleConfig getModuleConfig(DOMString moduleName);

void updateModuleConfig(DOMString moduleName, ModuleConfig config);

// Service Configuration

ServiceConfig getServiceConfig(DOMString serviceName);

void updateServiceConfig(DOMString serviceName,

ServiceConfig config);

};

### getConfig()

Returns the global configuration object.

**Parameters:**

(none)

**Returns:**

The global configuration object.

### updateConfig()

Assigns the global configuration for the system. This is first called when Core.init() is called and also may be called later if new configuration settings are returned from the server. After initial configuration is set, all further calls are assumed to be diffs of settings that should be changed rather than an entirely new configuration object.

**Parameters:**

* config - the global configuration object

**Returns:**

(nothing)

### getCoreConfig()

Returns the configuration object for the Core.

**Parameters:**

(none)

**Returns:**

The Core configuration object.

### updateCoreConfig()

Assigns the configuration for the named core. All calls are assumed to be diffs of settings that should be changed rather than an entirely new configuration object.

**Parameters:**

* config - a partial or complete Core configuration object

**Returns:**

(nothing)

### getModuleConfig()

Returns the configuration object for a given module.

**Parameters:**

* moduleName - the name of the module to retrieve configuration information for.

**Returns:**

The module configuration object or null if the named module doesn’t exist.

### updateModuleConfig()

Assigns the configuration for the named module. All calls are assumed to be diffs of settings that should be changed rather than an entirely new configuration object.

**Parameters:**

* moduleName - the name of the module to update configuration information for.
* config - a partial or complete module configuration object

**Returns:**

(nothing)

### getServiceConfig()

Returns the configuration object for a given service.

**Parameters:**

* serviceName - the name of the service to retrieve configuration information for.

**Returns:**

The service configuration object or null if the named service doesn’t exist.

### updateServiceConfig()

Assigns the configuration for the named service. All calls are assumed to be diffs of settings that should be changed rather than an entirely new configuration object.

**Parameters:**

* serviceName - the name of the service to update configuration information for.
* config - a partial or complete service configuration object

**Returns:**

(nothing)

## CoreConfig

The CoreConfig object represents configuration data used by the Core.

dictionary CoreConfig {

String moduleBase;

ModuleData modules;

};

### moduleBase

The base URL from which new modules should be loaded. This is used for dynamically loading new modules. The constructed URL is moduleBase + moduleName + ".js". Therefore, the value for moduleBase should end with a / character.

### modules

Any object containing information about the modules registered with the system. Each key in ModuleData is the name of a module, and each module has some information associated with it in the following form:

{

modules: {

moduleName1: {

enabled: false,

events: [ "click", "mousedown" ]

},

moduleName2: {

enabled: true,

events: [ "keydown", "keypress" ]

},

}

}

The enabled key indicates if the modules should be automatically started or not. If this key is omitted, then enabled is assumed to be true. The events array contains all of the events that a module is interested in receiving.

# Use Cases

## Boostrapping

To start the system, the following steps must occur:

1. Include the Core.
2. Include ConfigService.
3. Include all other services.
4. Include relevant modules.
5. Pass GlobalConfig into Core.init().

## Core Initialization (Core.init())

The Core initialization process is as follows:

1. Create ConfigService by calling its ServiceCreator.
2. Pass GlobalConfig to ConfigService.
3. Subscribe to the configupdated event from ConfigService.
4. Create each remaining service for the first time.
5. Initialize each module according to their configuration information, starting the appropriate modules by:
   1. Creating a unique ModuleContext object for each module.
   2. Passing that ModuleContext object to the ModuleCreator for each module.
   3. Storing the returned Module object so it can be accessed later.
   4. Calling Module::init() on each module if it is enabled in configuration.
6. Subscribe to each necessary DOM event for modules using the BrowserService.
7. Set an internal flag to indicate that the Core has been initialized.

## Event Flow

Modules receive event notifications from the Core via the following steps:

1. Core.init() reads all module information and assigns event handlers for all DOM events requested by modules.
2. When an event occurs, the Core reads module configuration to determine which modules are interested in that event.
3. The WebEvent object is passed to Module::onevent() on each module that is enabled and interested in the event.
4. The module can inspect the WebEvent object to process it appropriately.

For instance, if both Replay and Usability modules are interested in the click event, the following steps occur:

1. When Core.init() is called, it reads the module configuration information and sees there are two modules interested in the click event. A single event handler is then assigned to click on the document using the BrowserService.
2. When the click event handler is triggered by a user click, the Core event handler is called, receiving a WebEvent object and then reads configuration to see which modules are interested in the click event (configuration may have changed since the last time an event occurred, so it’s important to read the configuration again).
3. Having found that Replay and Usability are interested in click:
   1. Check that the Replay module is started.
   2. If Replay is started, then the Replay module’s Module::onevent() method is called and the WebEvent object is passed in.
   3. Check that the Usability module is started.
   4. If Usability is started, the Usability module’s Module::onevent() method is called and the same WebEvent object is passed in.

## Dynamic Configuration Change

Which modules are interested in a single event may change during the page lifecycle. The configuration may be updated dynamically at some point, or another module may be loaded from the server. In either case, the following steps are occur:

1. ConfigService receives new configuration data.
2. ConfigService fires the configupdated event.
3. The Core receives the configupdated event and re-initializes its DOM event handlers, removing any that are no longer needed and added new events that weren’t previously covered. It also checks to see if any modules need to be stopped or started.
4. The next time an event occurs, the appropriate modules are notified because this information is read from configuration each time.

## Dynamic Module Loading

A module may be loaded at any time during the lifecycle of the page. This occurs by:

1. The ConfigService receives a configuration update from the server.
2. The ConfigService fires the configupdated event.
3. The Core receives the configupdated event and inspects the configuration to determine if any new modules are being requested.
4. If the Core finds a new module, it makes a request to load that module’s JavaScript.
5. When the module JavaScript loads, it contains a call to Core.addModule(). This method checks the internal initialization flag and since the Core has already been initialized, it immediately invokes the new module’s ModuleCreator (Step 5 in Core Initialization) to create the new module.

# Example Code

This section contains sample code skeletons for implementing various interfaces for the UI SDK architecture. In these examples, the Core interface is implemented by the TLT object.

## Module Definition

A module is defined by calling TLT.addModule() and passing in the module name and a function that is used to construct the module (a ModuleCreator). That function is passed a ModuleContext object, which allows the module to interact with the Core indirectly.

TLT.addModule("name", function(context) {

// initialization

return {

// Lifecycle Methods

init: function() {},

destroy: function() {},

// Events and Messaging

onevent: function() {},

onmessage: function() {}

};

});

## Service Definition

A service is defined by calling TLT.addService() and passing in the service name and a function that is used to construct the module (a ServiceCreator). That function is passed a ServiceContext object, which allows the module to interact with the Core indirectly.

TLT.addService("name", function(context) {

// initialization

return {

// Service interface

};

});

## Module Listening for DOM Events

This module is listening for several DOM events and performs different tasks depending on the event. It counts mousedown events locally and reports after 10 have occurred and also reports click events immediately.

TLT.addModule("name", function(context) {

var mouseDownCount = 0;

return {

init: function() {

// in case the module is stopped and then restarted

mouseDownCount = 0;

},

onevent: function(event) {

if (event.type == "click") {

context.post({

type: "click"

});

} else if (event.type == "mouse" &&

event.subType == "mousedown") {

mouseDownCount += 1;

if (mouseDownCount == 10) {

context.post({

type: "mousedown10"

});

}

}

}

};

});

# Recommendations & Changes

The following list is a summary of recommendations and changes reflected in this architecture design.

* **Remove AMD support as a requirement for the first release.** The primary benefit AMD provides is the ability to calculate dependencies on the fly. Tealeaf doesn’t need this capability due to the standalone nature of modules. As such, it is more efficient to create your own loading mechanism. If AMD is needed in the future, it is easily added as a wrapper on this architecture.
* **Standardize definition of “module”.** In the original design, everything was a module, having the same standing in the system. However, not everything behaves the same way. The proposed changes define two types of objects: services and modules. Services are utility objects that provide some functionality to the Core while modules have no direct knowledge of the Core or the rest of the system.
* **Introduce further abstractions.** This architecture design introduces some further abstractions to enable easier testing:
  + ModuleContext abstracts the Core away from modules.
  + ServiceContext abstracts the Core away from services.
  + The Core now only manages modules and services.
  + TestCore is an object used purely for testing modules and services.
* **Standardize module interfaces.** Instead of having custom events that modules must listen for, trigger specific methods on the modules when events occur. That way, the trigger events can change easily without module code changing. The init(), destroy(), onevent(), and onmessage() methods handle this.
* **Remove Lifecycle.** The Lifecycle module in the original architecture isn’t actually a module, it’s just a bootstrap routine. This can be abstracted out as a small script that is appended to the overall file to start it up rather than being a full-fledged module.