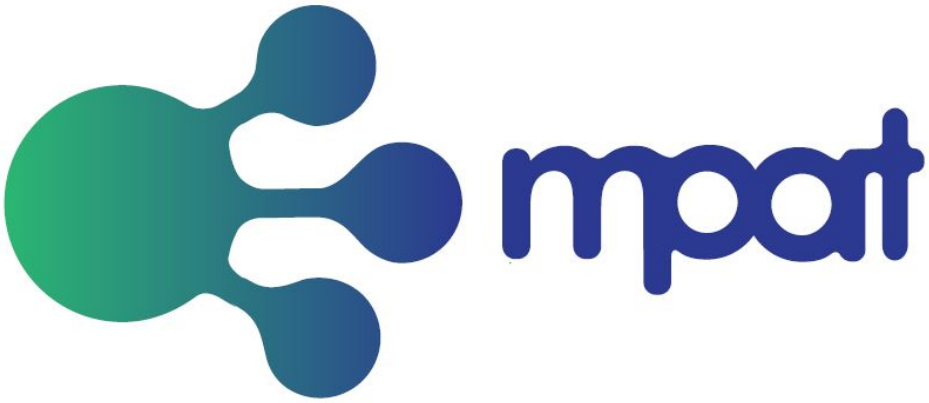


D4.6 Contributors Guidelines

Deliverable D4.6

MPAT File ID:	D4.6_Contributor_Guidelines.docx
Version:	1.0
Deliverable number:	D4.6
Authors:	Stefano Miccoli (Fincons), Marco Ferrari (Fincons), Christian Fuhrhop (FOKUS), Jean-Claude Dufourd (Paristech), Benedikt Vogel (IRT), Jamie Jellicoe (ULANC)

Work Package:	WP4
Task:	T4.3
Nature:	R – Report
Dissemination:	PU – Public
Status:	Final
Delivery date:	30.11.2017

Version and controls:

Version	Date	Reason for change	Editor
0.1	15.11.2017	Initial working version	Stefano Miccoli
0.2	22.11.2017	Fincons inputs	Stefano Miccoli, Marco Ferrari
0.3	24.11.2017	Paristech & Fokus inputs	Jean-Claude Dufourd, Christian Fuhrhop
0.4	26.11.2017	IRT inputs	Benedikt Vogel
0.5	28.11.2017	Lancaster inputs	Jamie Jellicoe
1.0	30.11.2017	Final submitted version	Stefano Miccoli

Acknowledgement: The research leading to these results has received funding from the European Union's Horizon 2020 Programme (H2020-ICT-2015, call ICT-19-2015) under grant agreement n° 687921.

Executive summary

As part of the open source strategy, this deliverable is addressed to developers who have not been part of the MPAT consortium in order to en thought to clarify all steps required steps to start specifies the information required by non-insider developers to start working with MPAT and produce custom components and functionalities.

It is structured in six sections, excluding introduction one, that describe different steps in the development lifecycle.

The first section, “Configuration of the development environment”, provides the information needed to setup a local environment to start playing with MPAT and implementing features.

The second section, “Configuration of the deployment infrastructure”, is oriented to operations rather than developments and suggests one possible solution for setting up a professional deployment workflow.

The third section, “Working with MPAT core” is the pillar of the document and include all the information needed to develop new features and customize MPAT.

The fourth section, “Working with localization (i18n)”, tackles a relevant issue when it comes to adopt MPAT in an heterogeneous area in terms of spoken language. Since the first professional workshop it was clear that even if widely adopted, english could not be the only language for the MPAT interface. This section explain how to use localization and how to add languages other than the native ones of the MPAT consortium members.

The fifth section, “Working with analytics”, elaborates another relevant topic in the era of big data and consequent profiling, namely the tracking of end-user behaviour and application usage.

The final section, “API Reference”, aims to recap the methods and functions of the APIs that are more likely to be used by third parties in the typical scenarios of implementation of new functionalities as well as integration with tracking systems part of the Broadcaster infrastructure.

Table of Contents

Introduction

This document contains the contributor guidelines for MPAT.

Most of the guidelines contained in the document are technical guidelines, describing how to add components and plug-ins to the MPAT code to stay in line with previous developments and to avoid breaking working code.

In addition to these more technical guidelines, any project with multiple collaborators also requires guidelines that cover more general aspects, covering issues such as code style, design philosophy and legal matters.

As we have based MPAT on WordPress, for these issues we primarily refer to the WordPress guidelines and best practices, which can be found here:

https://make.wordpress.org/core/handbook/

The most relevant part in regard to MPAT is the best practices section at:

https://make.wordpress.org/core/handbook/best-practices/

For components based on react, developers should also note these best practice guidelines

https://engineering.musefind.com/our-best-practices-for-writing-react-components-dec3eb5c3fc8:

We expect developers to follow the WordPress and React rules and guidelines where appropriate and will not repeat them in this document.

Beginning of this document describe the recommended development set-up.

Based on that, the subsequent sections describe how to add or modify elements related to the MPAT core.

Most broadcasters in Europe are operating on a national basis. HbbTV offerings are usually required to be provided in the local language. To make it easier for broadcasters to adapt MPAT to their local language requirements, a separate chapter describes the localization facilities of MPAT and gives guidelines on their utilization.

As already mentioned in this introductions, we mainly refer to the best practice guidelines of WordPress and React for the MPAT contributions, as most of them are directly applicable for MPAT and ensure that MPAT continues to work smoothly with established tools. The final section of this document, however, has some additional best practices for MPAT, which arise due to the fact that MPAT is specifically aiming at creating application for a TV environment, where some domain specific additional rules exist.

Configuration of the development environment

Setup of wordpress system

Although MPAT is delivered in the form of a set of wordpress plugins and a theme, when looking at the official GIT repository[1] the file structure might not look as a standard Wordpress layout. This is due to the decision of adopting the Bedrock[2] boilerplate which brings with it several benefits such as dependency management with Composer[3] and environment specific configuration. This setup is also optimized for the typical 3 or 4 tier deployment infrastructure (development, testing, staging, production) in use in professional environments. Additional coverage on deployment phase is described in the following chapter.

MPAT inherits dependencies from Wordpress[4] with further requirements for the React development. At an high level the requirements can be summarized as a LAMP/WAMP stack with inclusion of Node.js and NPM (not required but highly recommended to simplify the development).

Wordpress project can be cloned as GIT project from the public repository on github.

Required configuration steps are described in project's README.md and will not be repeated here.

Setup of webpack

As previously said, MPAT core is built with React.js with the support of Webpack together with Babel to take care of all the tedious tasks that involve the transpiling of code from ES6 to ES3, bundling of files and syntax checking.

Since custom plugins are independent bundles from the core, they have no visibility of ES6 modules from the core. Thus, writing custom React components would require to import the whole library again. In the same way, having multiple custom plugins would result in multiple, unnecessary, imports of React library, affecting the performances of the end-user applications, especially with slow network connections.

And here is where Webpack comes in aid. Leveraging on its library and external features[5], MPAT core is able to declare some modules as libraries, while custom plugins can reference those libraries as externals in their webpack configuration and then including them in components as ES6 modules.

Since webpack libraries are stored in JS global variables, developers could theoretically use them as object methods, however the following documentation assumes that Webpack is used in custom plugins too.

A `webpack.config.js` file that include core libraries can be written as follow:

```
var webpack = require('webpack');
var path = require('path');
var debug = process.env.NODE_ENV !== 'production';


const DebugDefines = {
    'DEBUG': 'true'
};
```

[1] Official GIT repository for MPAT project - https://github.com/MPAT-eu/MPAT-core
[2] Bedrock wordpress boilerplate - https://roots.io/bedrock/
[3] Composer dependency manager website - https://getcomposer.org/
[4] Wordpress requirements - https://wordpress.org/about/requirements/
[5] Webpack library & externals - http://webpack.github.io/docs/library-and-externals.html

```
const ProductionDefines = {
    'process.env': {
        'NODE_ENV': '"production"'
    },
    'DEBUG': 'false'
};

module.exports = {
    context: path.join(__dirname),
    entry: {
       backend: [...],
       frontend: [....],
    },
    devtool: debug ? "inline-sourcemap" : null,
    module: {
        loaders: [
            {
                test: /\.jsx?$/,
                include: path.join(__dirname, "js/src"),
                loader: 'babel-loader',
                query: {
                    presets: ['react', 'es2015', 'stage-0'],
                    plugins: ['react-html-attrs', 'transform-class-properties',
'transform-decorators-legacy'],
                }
            }
        ]
    },
    output: {
        path: __dirname + "/js/dist/",
        filename: "[name].min.js"
    },
    externals: {
        'react-backend': "mpat_admin.React",
        'react-frontend' : "mpat_core.React",
        'component-loader-backend' : "mpat_admin.componentAPI",
        'component-loader-frontend' : 'mpat_core.componentAPI',
        'keyBinding' : 'mpat_core.KeyBindingAPI',
        'analytics' : 'mpat_core.analyticsAPI',
        'admin-utils' : 'mpat_admin.utils',
        'core-utils' : 'mpat_core.utils',
        'application' : 'mpat_core.application'
    },
    plugins: debug ? [
        new webpack.DefinePlugin(DebugDefines)
    ] : [
        new webpack.optimize.DedupePlugin(),
        new webpack.optimize.OccurenceOrderPlugin(),
        //Make Uglify JS remove all the React debug messages
        new webpack.DefinePlugin(ProductionDefines),
        //see https://github.com/mishoo/UglifyJS2 for the parameter meanings
        new webpack.optimize.UglifyJsPlugin({
            compress:{
                warnings:false,
                drop_console: true
            },
        }),
    ]
};
```

Focusing on the externals section:

Library name	Description
`mpat_admin.React` and `mpat_core.React`	React library for backend and frontend
`mpat_admin.componentAPI` and `mpat_core.componentAPI`	APIs to interact with MPAT components, described in following dedicated section
`mpat_core.KeyBindingAPI`	APIs to interact with the TV remote control, described in following dedicated section
`mpat_core.analyticsAPI`	APIs to track application usage with external analytics systems
`mpat_admin.utils`	Helper classes for admin UI composition
`mpat_core.utils`	Helper classes for frontend UI composition
`mpat_core.application`	Container object for application settings and properties

Advantages of Babel transpiler

Babel is a tool that helps developers write code in the latest version of JavaScript. When the environment where the web application is executed, such as an outdated browser, does not support certain features natively, Babel helps the developer translating them in an understandable form through polyfills and prototypes definitions.

```
// ES2015 arrow function transpiling
// From
[1, 2, 3].map((n) => n + 1);
// To
[1, 2, 3].map(function(n) {
  return n + 1;
});
```

Babel is a crucial possibility to use modern Javascript, especially for hbbTV applications. The HbbTV Standard 1.0 and 1.5 requires ECMA Script Version 3 (ES3). HbbTV 2.0 refers to ES5, but devices that support HbbTV 2.0 do not have any market relevance to date[6].

Modern Javascript is ES6, so it is not possible to use modern Javascript HbbTV application without using a tool that translates the ES6 code into ES5 or ES3 code.

Developers can also avoid using Babel, but should be aware that the produced must be ES3 compliant to be sure that it is executed with no errors on HbbTV devices.

It's true that many devices support ES5 or even ES6, but by default they don't have to. Extensive tests within 2017 have repeatedly shown that about 13%-16% of devices do not support ES5, only ES3.

If you develop HbbTV applications that are not compatible with ES3, the applications will not work on 13-16% of devices in the market.

[6] November 2017

The above numbers refer to german market, an early adopter of HbbTV. Other considerations can be done based on different reference markets.

Configuration of the deployment infrastructure

When it comes to deploy a web application there are several paths that developers and operations can pursue. A fairly adopted solution is the so called 3(-4 based on needs) tier deployment infrastructure, where tiers refer to different environments used to develop features locally (development environment), tests them on the pre existing code and data (testing and/or staging environments) and finally publish the application to the world (production environment). HbbTV applications make no exception and so MPAT comes with preconfigured tools that simplify the deployment and disaster recovery processes.

MPAT uses Capistrano, a remote server automation and deployment tool written in Ruby. After having cloned the project, users are able to configure remote enviroments via configuration files that reside in `config\deploy` folder from the project root folder. In that folder users can find also a sample configuration file for Amazon AWS based environments.

It is worth mentioning that the use of capistrano brings additional dependencies, namely Ruby and Bundler. The former to execute the Capistrano scripts and the latter to automatically take care of dependencies and download Capistrano itself.

The exhaustive instructions to configure local and remote environents can be found in `deployment.md` file and so will not be repeated here.

Working with MPAT core

ComponentAPI: definition of components

Definition of backend component

After having imported the ComponentLoader from the core API, it can be used as follow in the custom component:

```
import { componentLoader } from '../ComponentLoader';
[...]
componentLoader.registerComponent('menu', {
        edit: MenuEdit,
        preview: MenuPreview
}, {
        isHotSpottable: false,
        isScrollable: true,
        hasNavigableGUI: true,
        isStylable: true,
        isComponentTrigger: true
        }, {
        navigable: true
        }
);
```

Where `MenuEdit` and `MenuPreview` refer to valid React component definition, such as stateless and presentational components, functions or classes that extend `React.Component`.

Definition of frontend component

Registration of frontend component follows the same rules of the backend one,

Definition of component views

MPAT offers the functionality of component views to allow the reuse of components, which share the same concept and data, but have sensibly different visual representations, that cannot be achieved with basic styling, i.e. different order of elements in page. An example for that could be a news component that retrieve information from external services. Although the meaningful information has the same nature among all services (i.e. news feed url, number of items, etc) the way that information is presented to the final user can be completely different. Registering different backend components may lead to a cluttered admin UI and increase the maintenance time when, for example, a new field, shared by all services has to be added.

Developers can add new frontend views for existing components, whether they are included in the core, in contributed plugins or defined by the developers themselves, as follow:

```
# in frontend component
componentLoader.registerFrontendView('menu', 'firstView', MenuFirstViewContent);
```

Where the first parameter refers to the component which the view will be registered to, the second refers to the name of the view and the third is the valid React component definition.

It is worth noting that even if there is no specific configuration of a view in the admin UI, it is still required to register it in the backend as well, to allow the users to choose it from the dedicated dropdown box in the component settings.

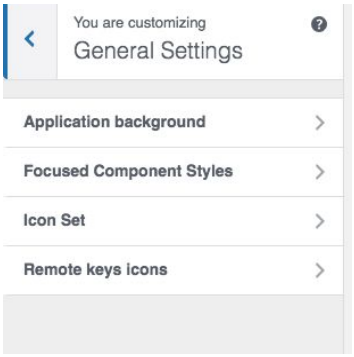
```
# in backend component
componentLoader.registerFrontendView('menu', 'firstView', {});
componentLoader.registerFrontendView('menu', 'secondView', {});
```

Since there is no view representation in the backend, the developers are not requested to specify the React component, empty object or `null` can be passed instead.

Overwrite and reuse of core components

Similar to the definition of frontend view, there is another way to have different frontend representation for an existing component that consist in the complete overwrite of a view, typically the frontend, with a custom implementation. It is worth noting that the the overwriting of the component applies to any site that has enabled the plugin containing the custom implementation. It is then clear that when compared to the frontend view solution, the overwrite leads to a cleaner admin UI at the cost of more fragile and less flexible solution.

Styling components



Default styles are set in the Customizer. Here you can find options for setting the site, page and component level settings.

The hierarchy for styling in MPAT starts with CSS classes, which are overridden by parent inline styles and then finally component inline styles.

The 'General Settings' menu item in the Customizer provides site level style settings like, which icon set to use, background colours and so on.

The styles applied from the Customizer are applied in the standard wordpress way in, functions.php, using actions and filters.

```
                        add_action('wp_head', 'mpat_customizer_css');

$wp_customize->add_control('launcher_font_style', array(
               'type' => 'select',
               'label' => 'Choose a Style for Launcher font:',
               'section' => 'mpat_launcher_settings',
               'choices' => array(
                             'bolder' => 'Bolder',
                             'bold' => 'Bold',
                             'normal' => 'Normal',
                             'lighter' => 'Lighter',
               ),
));
```

Page level styles and component level styles can be overridden in the React backend interface using the WYSIWYG editor.

Although third parties cannot change or create themes they are able to change the default style behaviour via their pluggins.

Styles applied in the Customizer are rendered as hierarchical classes in the frontend with further customisation applied by React in the form of inline styles. React does this using the 'className' parameter.

Customization of user roles and capabilities

Being the outcome of a R&D project, MPAT presents only a minimal separation of user roles but further capabilities can be defined to fulfill professional needs. User management is applicable only to the admin UI, where two roles are present:

- Administrator
- Editor

There is also a third role, inherited by Wordpress multisite setup, which is Super Administrator.

By default, MPAT enforces the following capabilities to the two aforementioned roles when new sites are created. Any change can be applied programmatically[7] or with user management plugins such as Members[8] and User Role Editor[9].

Capabilities for Administration role

Capability	Description
create_pages	Allows access to Pages option in administration panel and add new pages
edit_pages	Allow to edit pages
edit_others_pages	Allows to edit pages created by others
delete_pages	Allows to delete pages
manage_mpat_options	Allows access to MPAT custom options in administration panel

Capabilities for Editor role

Capability	Description
create_pages	Allows access to Pages option in administration panel and add new pages
edit_pages	Allow to edit pages
edit_others_pages	Allows to edit pages created by others
delete_pages	Allows to delete pages
edit_theme_options	Allows access to Administration Panel option: · Appearance → Customize
manage_mpat_options	Allows access to MPAT custom options in administration panel

[7] Wordpress user management documentation - https://codex.wordpress.org/Roles_and_Capabilities
[8] https://it.wordpress.org/plugins/members/
[9] https://it.wordpress.org/plugins/user-role-editor/

Interaction with core navigation models

Frontend and backend share a common data format. The backend generates and edits the data. The frontend reads and interprets the data. They thus have to be synchronized.

This is true for the TimeLine part. Any extension of the TimeLine editor, which is specified in `mpat-plugins/react_src/backend/timeline`, has to be reflected in `mpat-plugins/react_src/frontend/ TimeLineController.jsx`.

Working with localization (i18n)

Localization is built into the current build process. When you recompile the JSX files with webpack, localization files are regenerated too.

Existing localization

Localization exists in two places.

In `mpat-plugins`, the languages directory contains Wordpress-style localization. See Wordpress documentation for this part.

In `mpat-plugins/react_src`, the localization directory contains the js-specific localization.

Definition & registration of additional languages

To add a new language to MPAT, edit `mpat-plugins/react_src/localization/i18n.js` to add a new line in the const i18n, in this manner (adding language xx)

```
const i18n = new LocalizedStrings({
    en: en,
    de: de,
    es: es,
    fi: fi,
    fr: fr,
    it: it,
    nl: nl,
    xx: xx
});
```

Then edit `mpat-plugins/react_src/localization/merge/buildall.js` to add lines such as:

```
const xx = require('./org/xx');
[...]
m = new Merge2File('xx', xx, JSON.parse(org), exppath);
m.save();
```

Finally, copy `mpat-plugins/react_src/localization/merge/org/en.js` to xx.js and edit this file to include translations for all messages.

Contribution to existing localization

When adding new text to be localized, developer need to consider the following. Messages for each language for the whole MPAT application are organized in one object per language. To add a message, you need to add the same text property to all language objects of `mpat-plugins/react_src/localization/merge/org/??.js`.

A message can be added in its own new section, or in one of the existing sections, at any level.

When defining a new message that is supposed to be used as `i18n.section7.message2`, it has to be appended object to each language object as follow:

```
section7: {
 message2: 'text of message2'
},
```

Use of localization APIs

To use localized messages:

- import constants.js with a line such as
  `import Constants from '../../../constants';`
- define the i18n object with a line such as
  `const i18n = Constants.locstr.section7;`
- then use i18n.message2 to refer to the localized string.

The language used in the MPAT interface is set in the MPAT Settings.

Working with analytics

Nowadays there are countless analytics solutions, from open source to proprietary ones.

For this reason, MPAT provides a convenient set of APIs to track user behaviour and interactions with HbbTV applications rather than binding with a specific solution. Integration with the open source solution Piwik, provided out-of-the-box by MPAT is an example of such integration and can be used as a reference when integrating with other systems.

The integration comes in two steps. The former require components to hook to the core abstraction layer, while the latter allows third parties to translate the abstract information like page views and component interactions to the specific implementation in the analytics system of choice.

MPAT comes with an out-of-the-box integration with the open source solution Piwik[10]. It can be used as an example for further integrations.

Binding components to core APIs

When it comes to write custom components, it is recommended to specify all the interactions that are worth to be tracked. To do so, developers are required to import the tracking object exposed by MPAT core.

```
# in webpack.config.js (if used)
[...]
externals: {
        [...],
        'analytics' : 'mpat_core.analyticsAPI',
        [...]
},
[...]
```

If custom plugin does not use Webpack (highly recommended), analytics methods are still accessible from the mpat_core global JS variable, as follow:

```
# track page view
window.mpat_core.analyticsAPI.trackPageview(pageUrl, pageTitle = '')
# track user action
window.mpat_core.analyticsAPI.trackAction(category, action, name = '', value = '')
```

The previous code allows frontend components to include the tracking methods from the analytics object.

```
# in frontend component
import { trackAction } from 'analytics';
[...]
onArticleExit() {
    trackAction('blogpost', 'percentageRead', articleTitle, progress);
}
```

For reference documentation on tracking methods please refer to the API reference section.

[10] Piwik official site - https://piwik.org/

Integrating analytics systems

MPAT analytics APIs are based on the simple principle of the Event Listener. This makes very easy for third parties to integrate any existing web analytics solution in MPAT.

Depending on the degree of abstraction is requested, developers might want to expose some specific fields to configure the tracking service.

While it is possible to define a custom admin UI page following Wordpress guidelines[11], it is also possible to hook to the dedicated page (in admin sidebar MPAT → Analytics) defined by the core, having a cleaner layout in the application settings.

To add custom fields to the Analytics settings page, developers are required to hook to `admin_init` standard action to register a new settings section as follow (being `NewAnalytics` a sample name):

```
<?php
namespace MPAT\Settings;

use MPAT\Settings\FormHelper;

class NewAnalytics {
  // name of analytics implementation (used by wp to properly store values in db)
  const OPTION_NAME = "new_analytics_params";

 // method has to be triggered at plugin bootstrap
  static function register_analytics() {
   // hook to admin_init to register new settings
    add_action("admin_init", array(self::class, "register_settings"));
   // hook to mpat_analytics_variables to set additional values to the analytics group in
frontend
   add_filter("mpat_analytics_variables", array(self::class,
"register_frontend_variables"));
   // hook to add specific javascript inline (required by most of analytics systems to
initialize themselves)
    add_action("mpat_analytics_scripts", array(self::class, "print_config_script"));
  }
            //
  static function register_settings() {

    register_setting( 'mpat-analytics', self::OPTION_NAME );

    add_settings_section(
      'new_analytics_config',
      __( 'New analytics configuration', 'mpat' ),
```

[11] Wordpress official guidelines for settings page https://codex.wordpress.org/Creating_Options_Pages

```
    array(),
    'mpat-analytics'
  );

 // Checkbox to toggle analytics
  add_settings_field(
    'new_analytics_enabled',
    __( 'Enable New Analytics tracking', 'mpat' ),
    array(FormHelper::class, "checkbox_field_render"),
    'mpat-analytics',
    'new_analytics_config',
    array(
        "option_name" => self::OPTION_NAME,
        "field_name" => "enable",
        "default_value" => false
    )
  );

 // Text to store account key
  add_settings_field(
    'new_analytics_key',
    __( 'New analytics account key', 'mpat' ),
    array(FormHelper::class, "text_field_render"),
    'mpat-analytics',
    'new_analytics_config',
    array(
        "option_name" => self::OPTION_NAME,
        "field_name" => "key",
        "default_value" => false
    )
  );
}

static function get_config() {
  return get_option(self::OPTION_NAME, array(
      "enable" => false,
      "url" => null,
      "siteid" => null,
  ));
}
static function register_frontend_variables($systems) {
  $config = self::get_config();
```

```
    if ($config["enable"]) {
      $systems["new_analytics"] = $config;
    }
    return $systems;
  }

  static function print_config_script() {
    $config = self::get_config();

    if ($config["enable"]) {
      ?>
       /* include here the tracking snippet */
     <?php
    }
  }
}
```

It is worth noting that `mpat_analytics_variables` is a filter rather than an action where the first parameter contains the data of every register analytics system registered in the MPAT instance. It is then crucial that whether or not it is changed, it has to be the return value of the callback function, to avoid any side effect on other systems.

Moreover, the `NewAnalytics` class is using the `FormHelper` class from MPAT core, while not mandatory, it is a convenient way to generate consistent HTML code for custom fields.

Next step is to hook to javascript events triggered by the MPAT core when a navigation model or component wants to track an action or page view.

```
export default class piwik {
 constructor() {
   window.addEventListener('onMPATpageview', this.trackPageview);
   window.addEventListener('onMPATevent', this.trackAction);
 }
 trackAction(eventHandler) {
   // eventHandler = {detail: {page:"", category: "", action: "", name: "", value: ""}})
   if (typeof _paq !== 'undefined') {...}
 }
 trackPageview(eventHandler) {
   // eventHandler = {detail: {page:"", category: "", action: "", name: "", value: ""}})
   if (typeof _paq !== 'undefined') {...}
 }
}
```

As shown in the snippet above, both events come with a parameter that holds all the information related to the event. In case of a page view tracking, the object has the follow structure:

```
{ detail: {
    page: "",  // url of the page just loaded
    title: "", // human-readable title of the page
} }
```

The object sent for other events is slightly more complex due to the required flexibility

```
{ detail: {
    page: "", // url of the current page
    category: "", // type of event, typically the name of the component that triggered the
event
    action: "",  // description of the interaction that triggered the event
    name: "",  // additional field to better describe the event
    value: "" // additional field to better describe the event
} }
```

API Reference

Component Loader

Method	Parameters	Description
`registerComponent(type, classes, options = {}, defaults = {}, styles = {})`	• type: component name • classes: object of valid React components (functions, ES6 classes, DOM elements…) • *options*: configuration of component behavior • *defaults*: default values for options • *styles*: customisable styles	Register a new component in MPAT system. Being admin and frontend UI completely decoupled, every component has to be registered in both frontend and backend module. The former to let users pick it from the UI when editing pages, the latter to have a visual representation
`unregisterComponent(type)`	• type: component name	Unregister a component
`overwriteComponent(type, newClasses)`	• type: component name • newClasses: object of valid React components (functions, ES6 classes, DOM elements…)	Replace representations (view, preview, edit) for a given component. Useful for replacing components defined in core or other plugins
`getComponents()`	• type: component name	Get all the registered components
`getComponentByType(type)`	• type: component name	Get a component by its name
`getComponentProperties(type)`	• type: component name	Get all the frontend specific properties of a component by its name
`getComponentProperty(type, propertyName)`	• type: component name • propertyName: name of a specific property	Get a property of a component by its name
`getComponentOptions(type)`	• type: component name	Get all the options of a component by its name
`getComponentOption(type, optionName)`	• type: component name • optionName: name of a specific option	Get a option of a component by its name
`getComponentDefaults(type)`	• type: component name	Get default values of a component
`getComponentDefault(type, name)`	• type: component name • name: name of the option	Get a default value of a component

`getComponentStyles(type)`	• type: component name	Get the definition of all the custom styles
`renderComponent(type, element, params)`	• type: component name • element: representation name • params: component data	Get a component representation and render it. Used by the core, very unlikely to be used in custom compoennts
`registerFrontendView(type, label, viewClass)`	• type: component name • label: name of the view • viewClass: valid react component	Add a frontend view to an already registered component
`getFrontendViews(type)`	• type: component name	Get all the frontend view for a registered component
`getFrontendView(type, view = 'default')`	• type: component name • view: name of the view	Get a specific frontend representation for a component

Optional parameters are written in *italic*

Concerning `classes` and `newClasses` parameters, both are structured as plain object where the key specifies the representation of a component and can be one of `view`, `preview` and `edit`. Classes (object) – component definition, all standard react solutions (namely functions, React.createClass() factory and ES6 classes) are supported, while ES6 classes are recommended for readability and uniformity. Three properties are managed:. Each of them contains the definition for the respective representation of the MPAT component.

Analytics

Methods used in components to trigger tracking

Method	Parameters	Description
`trackAction(category, action, name = '', value = '')`	• category: type of event, typically the name of the component, e.g. “blog” • action: the user action to be tracked, eg. “scrollArticles” • *name*: additional context info, e.g. “economics” • *value*: additional context info, e.g. “page 2”	Track an interaction with the application directly or indirectly triggered by the final user. Direct action: user scroll to next page Indirect action: article percentage read when going back to articles list.
`trackPageview(url, title = '')`	• url: url of the rendered page • *title*: human-readable title of the page	Track the load of a new page. By default, all the navigation models provided by the core track such event.

Optional parameters are written in *italic*

Glossary

ESx (3, 5…)	ECMAscript
JS	Javascript
CSS	Cascading Style Sheets
R&D	Research and Development
WP	Wordpress

Partner Short Names

Short Name	Name
FRAUNHOFER	Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung e.V. (DE)
IRT	Institut für Rundfunktechnik GmbH (DE)
RBB	Rundfunk Berlin-Brandenburg (DE)
ULANC	Lancaster University (UK)
MEDIASET	Reti Televisive Italiane S.p.A. (IT)
LEADIN	Leadin Oy (FI)
FINCONS	Fincons SpA (IT)
IMT	Institut Mines-Telecom (FR)