Mashroom Server

Version: **2.7.1**

https://www.mashroom-server.com

Table of contents

About

Mashroom Server is a *Node.js* based **Microfrontend Integration Platform**. It supports the integration of *Express* webapps on the server side and composing pages from multiple *Single Page Applications* on the client side (Browser). It also provides common infrastructure such as security, communication (publish/subscribe), theming, i18n, storage, and logging out of the box and supports custom middleware and services via plugins.

Mashroom Server allows it to implemented SPAs (and express webapps) completely independent and without a vendor lock-in, and to use it on arbitrary pages with different configurations and even multiple times on the same page. It also allows it to restrict the access to resources (Pages, Apps) based on user roles.

From a technical point of view the core of *Mashroom Server* is a plugin loader that scans npm packages for plugin definitions (package.json, mashroom.json) and loads them at runtime. Such a plugin could be an *Express* webapp or a *SPA* or more generally all kind of code it knows how to load, which is determined by the available plugin loaders. Plugin loaders itself are also just plugins, so it is possible to add any type of custom plugin type.

Key features

Portal

- Registration of Single Page Applications written with any frontend framework (basically you just need to implement a startup function and provide some metadata)
- Automatic registration of SPAs (**Remote Apps**) on remote servers or Kubernetes clusters (this allows independent life cycles and teams per SPA)
- Create static pages with registered SPAs (Apps) as building blocks
- Support for **dynamic cockpits** where Apps are loaded (and unloaded) based on some user interaction or search results
- Support for **composite Apps** which can use any registered SPA as building blocks (which again can serve as building blocks for other composite Apps)
- Each App receives a config object which can be different per instance and a number of JavaScript services (e.g. to connect to the message bus or to load other Apps)

- Support for **hybrid rendering** for both the Portal pages and SPAs (If an SPA supports server-side rendering the initial HTML can be incorporated into the initial HTML page. Navigating to another page dynamically replaces the SPAs in the content area via client side rendering)
- The App config can be edited via Admin Toolbar or a custom Editor App which again is just a plain SPA
- Client-side message bus for inter-app communication which can be extended to server-side messaging (to communicate with Apps in other browsers or even in 3rd party systems)
- Arbitrary (custom) layouts for pages
- Extensive **theming** support (Themes can be written in any Express template language)
- Support for multiple sites that can be mapped to virtual hosts
- Proxying of REST API calls to avoid CORS problems (HTTP, SSE, WebSocket)
- Support for global libraries that can be shared between multiple SPAs
- Delivering of Theme and Portal App resources via CDN
- Admin Toolbar to create pages and place Apps via Drag'n'Drop
- **Hot reload** of SPAs in development mode

Core

- Shared middlewares and services
- **Service abstractions** for security, internationalization, messaging, HTTP proxying, memory cache and storage
- Existing provider plugins for security (OpenID Connect, LDAP), storage (File, MongoDB), messaging (MQTT, AMQP) and caching (Redis)
- Integration of (existing) *Express* webapps

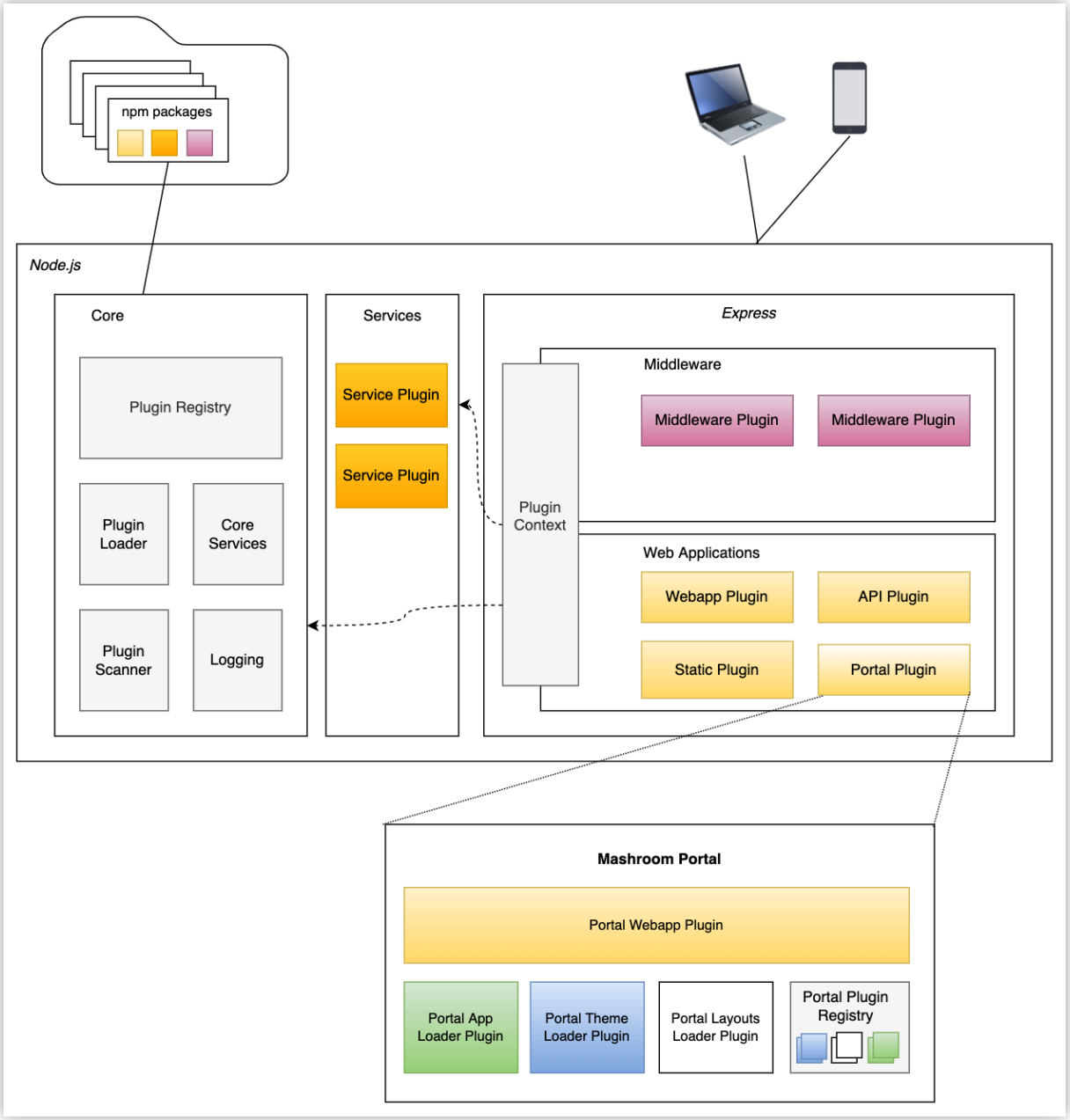
- Integration of (existing) *Express* (REST) APIs
- Role and IP based **access control** for URLs
- Definition of access restrictions for arbitrary resources (such as Sites, Pages, App instances)
- Single configuration file to override plugin defaults
- Support for **custom plugin types**
- Extensive **monitoring** and export in Prometheus format
- Hot deploy, undeploy and reload of all kind of plugins
- No compile or runtime dependencies to the server
- Fast and lightweight

Feature/Compatibility Matrix

	Supported
Operating Systems	Linux, MacOS, Windows
Node.js	18.x, 20.x, 22.x
HTTP	1.0, 1.1, 2 + TLS 1.1, 1.2, 1.3
Authentication	LDAP (Active Directory), OpenID Connect/OAuth2, local user database (JSON file)
Authorization	Role based; ACL (URL and HTTP method, based on roles and/or IP address); Resource permissions (Page, App instance, Topic, …)
Security	CSRF protection, Helmet integration
Storage	MongoDB, Plain JSON Files
Memory Cache	Local Memory, Redis
Messaging	MQTT (3.1, 3.1.1/4.0, 5.0), AMQP (1.0)
Session Storage	Local Memory (no Cluster support), shared Filesystem, Redis, MongoDB
API Proxy	HTTP, HTTPS, SSE, WebSocket
CDN	Any that can be configured as caching proxy
Clustering	yes (tested with PM2)

	Supported
Monitoring	CPU, Heap, Requests + Plugin Metrics; Exporter for Prometheus
Desktop Browsers	Chrome (latest), Firefox (latest), Safari (latest), Edge (latest)
Mobile Browsers	Chrome (latest), Safari (latest)

Architecture



Key concepts

Plugin definition

A plugin definition consists of two parts:

1. A plugin definition element, either in *package.json* or a separate *mashroom.json* file

2. A loader script (bootstrap)

A *package.json* with a *Mashroom Server* plugin definition looks like this:

```
{
    "$schema": "https://www.mashroom-server.com/schemas/mashroom-packagejson-extensi
    "name": "my-webapp",
    "version": "1.0.0",
    "dependencies": {
        "express": "4.16.4"
    },
    "devDependencies": {
    },
    "scripts": {
        "build": "babel src -d dist"
    },
    "mashroom": {
        "devModeBuildScript": "build",
        "plugins": [
            {
                "name": "My Webapp",
                "type": "web-app",
                "bootstrap": "./dist/mashroom-bootstrap.js",
                "requires": [
                    "A special service plugin"
                ],
                "defaultConfig": {
                    "path": "/my/webapp"
                }
            }
        ]
    }
}
```

The same in a separate *mashroom.json*

```
{
    "$schema": "https://www.mashroom-server.com/schemas/mashroom-plugins.json",
    "devModeBuildScript": "build",
    "plugins": [
        {
            "name": "My Webapp",
            "type": "web-app",
            "bootstrap": "./dist/mashroom-bootstrap.js",
            "requires": [
                "A special service plugin"
            ],
            "defaultConfig": {
                "path": "/my/webapp"
            }
        }
    ]
}
```

Multiple plugins can be defined within a single npm package.

The *type* element determines which plugin loader will be used to load the plugin. The optional *requires* defines plugins that must be loaded before this plugin can be loaded. The content *defaultConfig* differs per plugin type. It can be overwritten in the *plugins* section of the server configuration.

The *devModeBuildScript* property is optional. If present *npm run <devModeBuildScript>* is executed in *devMode* after every change in the package.

The bootstrap script for this case might look like this:

```
import webapp from './my-express-webapp';

import {MashroomWebAppPluginBootstrapFunction} from 'mashroom/type-definitions';

const bootstrap: MashroomWebAppPluginBootstrapFunction = async (pluginName, pluginCo
    return webapp;
};

export default bootstrap;
```

The context element allows access to the server configuration, the logger factory and all services.

Plugin context

The plugin context allows access to the logger factory und all services. The plugin context is available via:

- The pluginContextHolder argument in the bootstrap function
- The express request object (*req.pluginContext*)

Examples:

```
import type {MashroomLogger} from '@mashroom/mashroom/type-definitions';
import type {MashroomStorageService} from '@mashroom/mashroom-storage/type-definitic

const bootstrap: MashroomWebAppPluginBootstrapFunction = async (pluginName, pluginCo

    const pluginContext = pluginContextHolder.getPluginContext();
    const logger: MashroomLogger = pluginContext.loggerFactory('my.log.category');
    const storageService: MashroomStorageService = pluginContext.services.storage.se

        //...
};
```

```
app.get('/', (req, res) => {

    const pluginContext = req.pluginContext;
    const logger: MashroomLogger = pluginContext.loggerFactory('my.log.category');
    const storageService: MashroomStorageService = pluginContext.services.storage.se

    //...
});
```

NOTE: Never store the *pluginContext* outside a bootstrap or request handler because service references my change over time when plugins are reloaded. But it save to store the *pluginContextHolder* instance.

Setup

Minimum Requirements

- Node.js >= 14

Install

Just checkout the mashroom-portal-quickstart repo for a typical portal setup. Or mashroom-quickstart if you don't need the Portal plugin.

A single *package.json* is enough to set up a server instance. Plugins are just npm dependencies.

Configuration

The configuration files are expected in the folder where *mashroom* is executed. Alternatively you can pass the root folder as argument:

```
mashroom <path_to_config_files>
```

The following config files are loaded and merged together if present (in this order):

- mashroom.json
- mashroom.js
- mashroom.<node_env>.json
- mashroom.<node_env>.js
- mashroom.<hostname>.<node_env>.json
- mashroom.<hostname>.<node_env>.js
- mashroom.<hostname>.<node_env>.js

The typical configuration could look like this:

```
{
    "$schema": "https://www.mashroom-server.com/schemas/mashroom-server-config.json"
    "name": "Mashroom Test Server 1",
    "port": 8080,
    "indexPage": "/portal",
    "xPowerByHeader": "Mashroom Server",
    "tmpFolder": "/tmp",
    "pluginPackageFolders": [{
        "path": "./node_modules/@mashroom"
     }, {
        "path": "./my-plugin-packages",
        "watch": true,
        "devMode": true
    }],
    "ignorePlugins": [],
    "plugins": {
        "Mashroom Session Middleware": {
            "provider": "Mashroom Session Filestore Provider",
            "session": {
            }
        },
        "Mashroom Session Filestore Provider": {
            "path": "./data/sessions",
            "ttl": 1200
        },
        "Mashroom Security Services": {
            "provider": "Mashroom Security Simple Provider",
            "acl": "./acl.json"
        },
        "Mashroom Security Simple Provider": {
            "users": "./users.json",
            "loginPage": "/login"
        },
        "Mashroom Storage Services": {
            "provider": "Mashroom Storage Filestore Provider"
        },
        "Mashroom Storage Filestore Provider": {
            "dataFolder": "./data/storage"
        },
        "Mashroom Internationalization Services": {
            "availableLanguages": ["en", "de"],
            "defaultLanguage": "en"
        },
        "Mashroom Http Proxy Services": {
            "rejectUnauthorized": false,
            "poolMaxSockets": 10
        }
    }
}
```

The same as a Javascript file:

```
module.exports = {
    name: "Mashroom Test Server 1",
    port: 8080,
    indexPage: "/portal",
    xPowerByHeader: "Mashroom Server",
    tmpFolder: "/tmp",
    pluginPackageFolders: [{
         path: "./node_modules/@mashroom"
     }, {
        path: "./my-plugin-packages",
        devMode: true
    }],
    ignorePlugins: [],
    plugins: {

    }
}
```

Since version 1.3 the property values can also contain string templates and the environment variables are accessible via *env* object:

```
{
    "name": "${env.USER}'s Mashroom Server",
    "port": 5050
}
```

To enable HTTPS and/or HTTP2 you would add:

```
{
    "httpsPort": 5443,
    "tlsOptions": {
        "key": "./certs/key.pem",
        "cert": "./certs/cert.pem"
    },
    "enableHttp2": true
}
```

Properties

- *name*: The server name (default: Mashroom Server)
- *port*: The port the server should bind to (default: 5050)

- *httpsPort*: Additionally launch a HTTPS server on this port (requires *tlsOptions* as well)
- *tlsOptions*: Passed to Node TLS but the file paths (e.g. for "cert") are resolved relatively to the server config.
- *enableHttp2*: Enable HTTP/2 for the HTTPS server. If you enable this WebSockets will no longer work.
- *indexPage*: The start page if the root ('/') is requested (default: /)
- *xPowerByHeader*: The *x-powered-by* header to send; null disables the header (default: Mashroom Server)
- *tmpFolder*: The tmp folder for plugin builds and so on (default: OS specific temp dir)
- *pluginPackageFolders*: An array of folder paths that contain *Node.js* modules with *Mashroom Server* plugins An object in this array can have the following properties:
  - *path*: The plugin folder path, relative to the config file (mandatory)
  - *watch*: Determines if this folder should be watched for changed, new or removed packages (default: false)
  - *devMode*: If this is true the server automatically builds plugins on changes before reloading it. This option enforces *watch*. (default: false)
- *ignorePlugins*: An array of plugin names which shall be ignored (and not loaded)
- *plugins*: This section can be used to override the *defaultConfig* of arbitrary plugins

Security

To enable security you have to add the mashroom-security package and a provider package such as mashroom-security-provider-simple.

The security package provides access control lists based on URLs and a Service for managing and checking resource permissions manually.

> **NOTE**: The security in the Portal and for Portal Apps (SPA) is described below in *Mashroom Portal -> Security*.

ACL

You can secure every URL in *Mashroom* with the ACL, based on user role or IP. For example:

```
{
    "$schema": "https://www.mashroom-server.com/schemas/mashroom-security-acl.json",
    "/portal/**": {
        "*": {
            "allow": {
                "roles": ["Authenticated"]
            }
        }
    },
    "/mashroom/**": {
        "*": {
            "allow": {
                "roles": ["Administrator"],
                "ips": ["127.0.0.1", "::1"]
            }
        }
    }
}
```

NOTE: For more details check the mashroom-security documentation below.

Resource permissions

The *SecurityService* allows it to define and check resource permissions based on a *permission* key and the user roles. For example:

```
import type {MashroomSecurityService} from '@mashroom/mashroom-security/type-definit

export default async (req: Request, res: Response) => {
    const securityService: MashroomSecurityService = req.pluginContext.services.secu

    // Create a permission
    await securityService.updateResourcePermission(req, {
        type: 'Page',
        key: pageId,
        permissions: [{
            permissions: ['View'],
            roles: ['Role1', 'Role2']
        }]
    });

    // Check a permission
    const mayAccess = await securityService.checkResourcePermission(req, 'Page', pag
```

```
    // ...
}
```

This mechanism is used by the Portal for Site, Page and Portal App permissions.

> **NOTE**: For more details check the mashroom-security documentation below.

HTTP Security Headers

Use the mashroom-helmet plugin to add the Helmet middleware, which adds a bunch of HTTP headers to prevent XSS and other attacks.

CSRF

The mashroom-csrf plugin adds middleware that checks every POST, PUT and DELETE request for a CSRF token in the HTTP headers or the query. You need to use the *MashroomCSRFService* to get the current token.

> **NOTE**: The default Portal theme automatically adds the current CSRF token in a meta tag with the name *csrf-token*.

Logging

The logger is currently backed by log4js .

The configuration is expected to be in the same folder as *mashroom.cfg*. Possible config files are:

- log4js.<hostname>.<node_env>.js
- log4js.<hostname>.<node_env>.json
- log4js.<hostname>.js
- log4js.<hostname>.json
- log4js.<node_env>.js
- log4js.<node_env>.json
- log4js.js
- log4js.json

The first config file found from this list will be used. A file logger would be configured like this:

```
{
    "appenders": {
        "file1": {"type": "file", "filename": "log/mashroom.log", "maxLogSize":
        "file2": {
          "type": "file", "filename": "log/my-stuff.log", "maxLogSize": 10485760
          "layout":  {
            "type": "pattern",
            "pattern": "%d %p %X{sessionID} %X{clientIP} %X{username} %c - %m"
          }
        },
        "console": {"type": "console"}
    },
    "categories": {
        "default": {"appenders": ["file1", "console"], "level": "debug"},
        "my-stuff": {"appenders": ["file2"], "level": "info"}
    }
}
```

The following built in context properties can be used with %X{} or a custom layout:

- *clientIP*
- *browser* (e.g. Chrome, Firefox)
- *browserVersion*
- *os* (e.g. Windows)
- *sessionID* (if a session is available)
- *portalAppName* (if related to a Portal App)
- *portalAppVersion* (if related to a Portal App)
- *portalAppHost* (if related to a Remote Portal App)

You can use *logger.withContext()* or *logger.addContext()* to add context information.

For configuration details and possible appenders see log4js-node Homepage .

Logstash

To push the logs to logstash you can use the *logstash-http* package:

```
    "dependencies": {
      "@log4js-node/logstash-http": "^1.0.0"
    }
```

And configure log4js like this:

```
{
  "appenders": {
    "logstash": {
      "type": "@log4js-node/logstash-http",
      "url": "http://elasticsearch:9200/_bulk",
      "application": "your-index"
    }
  },
  "categories": {
    "default": {
      "appenders": [ "logstash" ],
      "level": "info"
    }
  }
}
```

Internationalization

The mashroom-i18n plugin provides a simple service to lookup messages on the file system based on the current user language.

You can use it like this from where ever the *pluginContext* is available:

```
import type {MashroomI18NService} from '@mashroom/mashroom-i18n/type-definitions';

export default (req: Request, res: Response) => {
    const i18nService: MashroomI18NService = req.pluginContext.services.i18n.service

    const currentLang = i18nService.getLanguage(req);
    const message =  i18nService.getMessage('username', 'de');
    // message will be 'Benutzernamen'

    // ...
}
```

One the client-side (in Portal Apps) you can use an arbitrary i18n framework (such as intl-messageformat). The current Portal locale will be passed to the Apps with the *portalAppSetup*:

```
const bootstrap: MashroomPortalAppPluginBootstrapFunction = (portalAppHostElement, p
    const { lang } = portalAppSetup;
    // lang will be 'en' or 'fr' or whatever
    const { messageBus, portalAppService } = clientServices;
    // ...
};
```

Caching

Mashroom tries to automatically use the most efficient caching mechanisms. All you need to do is to add the appropriate plugins.

Server-side

Add mashroom-memory-cache plugin and optionally mashroom-memory-cache-provider-redis if you want to use Redis instead of the local memory. Many other plugins (such as mashroom-storage and mashroom-portal) will automatically detect it and use it (see their documentation for more details).

Browser

The mashroom-browser-cache plugin provides a service to set *Cache-Control* headers based on a policy. For example:

```
import type {MashroomCacheControlService} from '@mashroom/mashroom-browser-cache/typ

export default async (req: Request, res: Response) => {

    const cacheControlService: MashroomCacheControlService = req.pluginContext.servi
    await cacheControlService.addCacheControlHeader('ONLY_FOR_ANONYMOUS_USERS', req,

    // ..
};
```

Other plugins (such as mashroom-portal) will automatically use it if present.

> **NOTE**: This plugin will always set *no-cache* header in *devMode*.

CDN

Since v2 *Mashroom* ships a CDN plugin that will automatically be used by mashroom-portal and other plugins to deliver static assets.

Basically, mashroom-cdn consists of a list of CDN hosts and a service to get the next host to use:

```
import type {MashroomCDNService} from '@mashroom/mashroom-cdn/type-definitions';

export default async (req: Request, res: Response) => {

    const cdnService: MashroomCDNService = req.pluginContext.services.cdn.service;

    const cdnHost = cdnService.getCDNHost();
    const resourceUrl = `${cdnHost}/<the-actual-path>`;

    // ..
};
```

NOTE: The mashroom-cdn plugin requires a CDN that works like a transparent proxy, which forwards an identical request to the *origin* (in this case Mashroom) if does not exist yet.

Virtual Host Path Mapping

The mashroom-vhost-path-mapper plugin can be used to map frontend paths to *internal* paths, based on virtual hosts.

So, lets say you want to map the *Mashroom Portal* site *site1*, reachable under http://localhost:5050/portal/site1, to www.my-company.com. In that case you would configure the plugin like this:

```
{
  "plugins": {
       "Mashroom VHost Path Mapper Middleware": {
           "hosts": {
              "www.my-company.com": {
                 "mapping": {
                    "/login": "/login",
                    "/": "/portal/site1"
                 }
              }
          }
       }
    }
}
```

If your *frontend base path* is different, e.g. www.my-company.com/foo, you would also have to set the *frontendBasePath* in the configuration.

NOTE: All other plugins will only deal with the rewritten paths, keep that in mind especially when defining ACLs.

Clustering

If you're going to run *Mashroom Server* in a cluster you should keep in mind:

- Use as session provider that can provide a shared session for all instances (such as *mashroom-session-provider-filestore*)
- If you use mashroom-messaging to communicate between browser windows you need to configure an external broker
- If you use mashroom-memory-cache you should configure Redis as shared cache server to maximize the hits
- If you use mashroom-storage (e.g. for the Portal) you either have to use Mongo as backend or to use a shared folder (e.g. SAN) for all instances
- If you use file log appenders you either have to make sure that your *log4js* config creates a file per node process, or you have to install *pm2-intercom* if you're using *pm2*. Read the *log4js* clustering documentation for details: https://github.com/log4js-node/log4js-node/blob/master/docs/clustering.md

A cluster safe log configuration could look like this:

```
const NODE_ID = process.env.pm_id || process.pid;

module.exports = {
    appenders: {
        file: {'type': 'file', 'filename': `log/mashroom.${NODE_ID}.log`, 'maxLogSiz
        console: {'type': 'console'}
    },
    categories: {
        default: {appenders: ['file', 'console'], 'level': 'info'}
    },
    disableClustering: true
};
```

Monitoring

Mashroom Server gathers a lot of internal metrics that can be exposed in different formats. Currently, there are two exporters available:

- for Prometheus
- for PM2 monitoring

To enable the metrics, just add *@mashroom/mashroom-monitoring-metrics-collector* and one of the exporter plugins.

The Prometheus metrics will be available at **/metrics**. An example Grafana Dashboard can be found in the Mashroom repo .

Here how it looks:

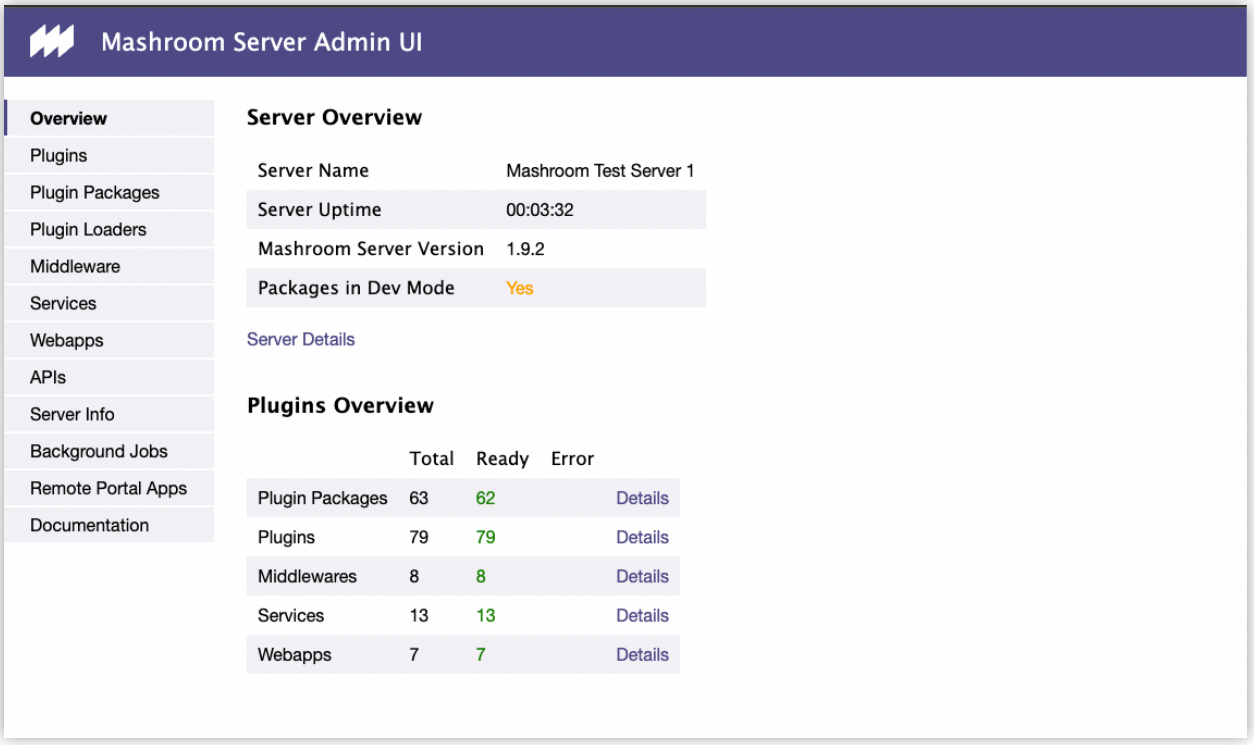
Health checks

There are a few integrated health checks available that can be used as probes for monitoring tools or to check the readiness/liveness of Kubernetes pods.

An overview of the health checks is available under http://<host>:<port>/mashroom/health

Admin UI

The *Mashroom Server* Admin UI is available under http://<host>:<port>/mashroom/admin

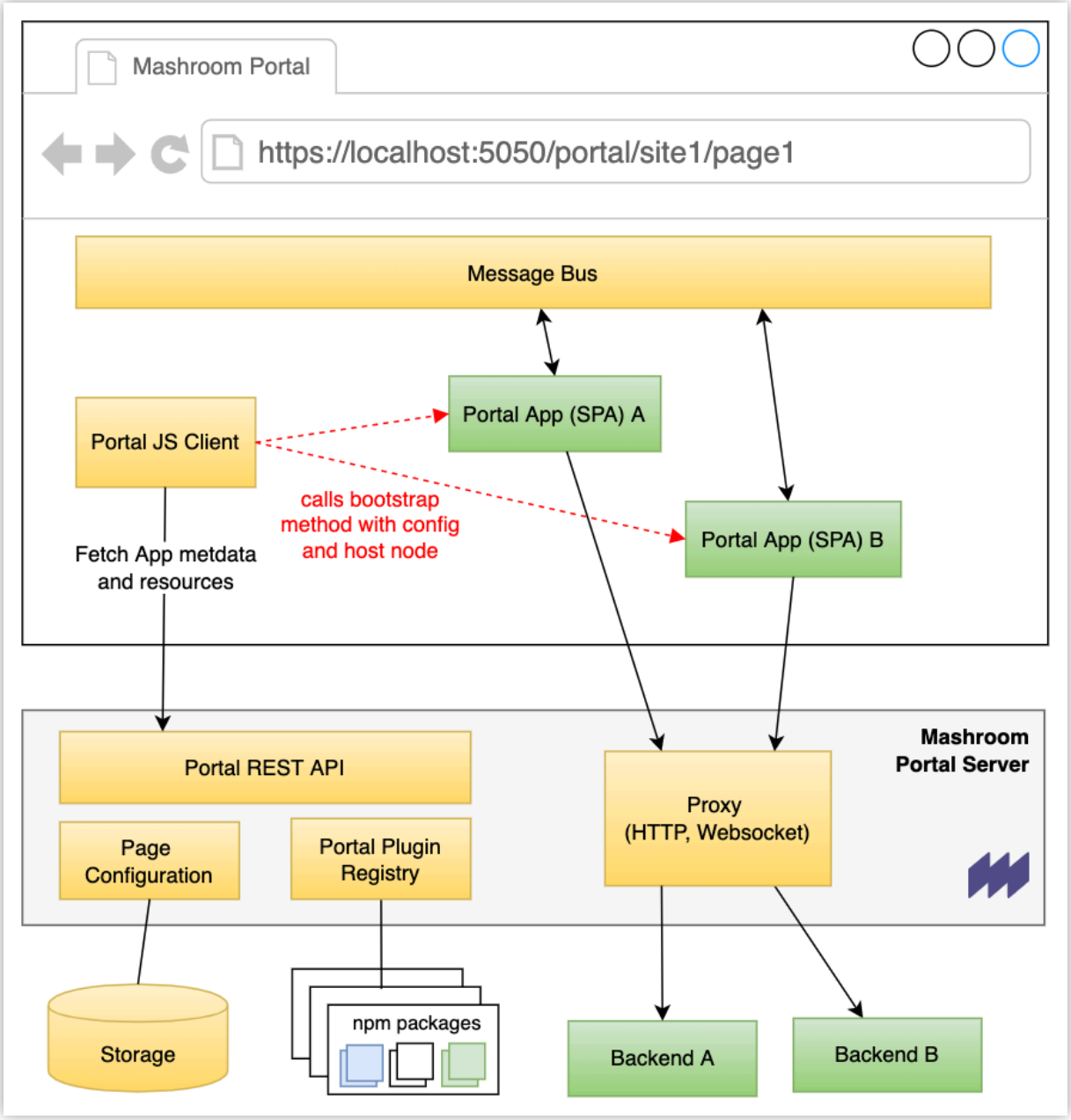


It contains:

- General server information
- Plugins with state and configuration
- Middleware stack (with order)
- All available services
- All loaded webapps
- Documentation

Mashroom Portal

Architecture



How does it work?

Every Portal App (SPA) has to expose a global *bootstrap* method. The *Portal Client* fetches the app metadata from the server, loads all required resources and calls then the *bootstrap* with the configuration object and the DOM element where the app should run in.

HTTP/Websocket Proxy

The built-in proxy allows the Portal App to access internal APIs. It supports HTTP, Websocket and SSE.

You have to define proxies like this in the plugin definition:

```
"defaultConfig": {
  "proxies": {
    "myApi": {
      "targetUri": "http://localhost:1234/api/v1"
    }
  }
}
```

To be able to use it in the boostrap of your App like this:

```
const bootstrap: MashroomPortalAppPluginBootstrapFunction = (portalAppHostElement, p
    const {lang, restProxyPaths} = portalAppSetup;

    fetch(`${restProxyPaths.myApi}/foo/bar?q=xxx`, {
        credentials: 'same-origin',
    }).then( /* ... */ );

    // ...
};
```

Proxy Interceptors

Plugins of type *http-proxy-interceptor* can be used to intercept all proxy calls and to check, enrich or manipulate requests or responses.

A typical use case is to add some security headers (e.g. Bearer) or do some redirect or caching.

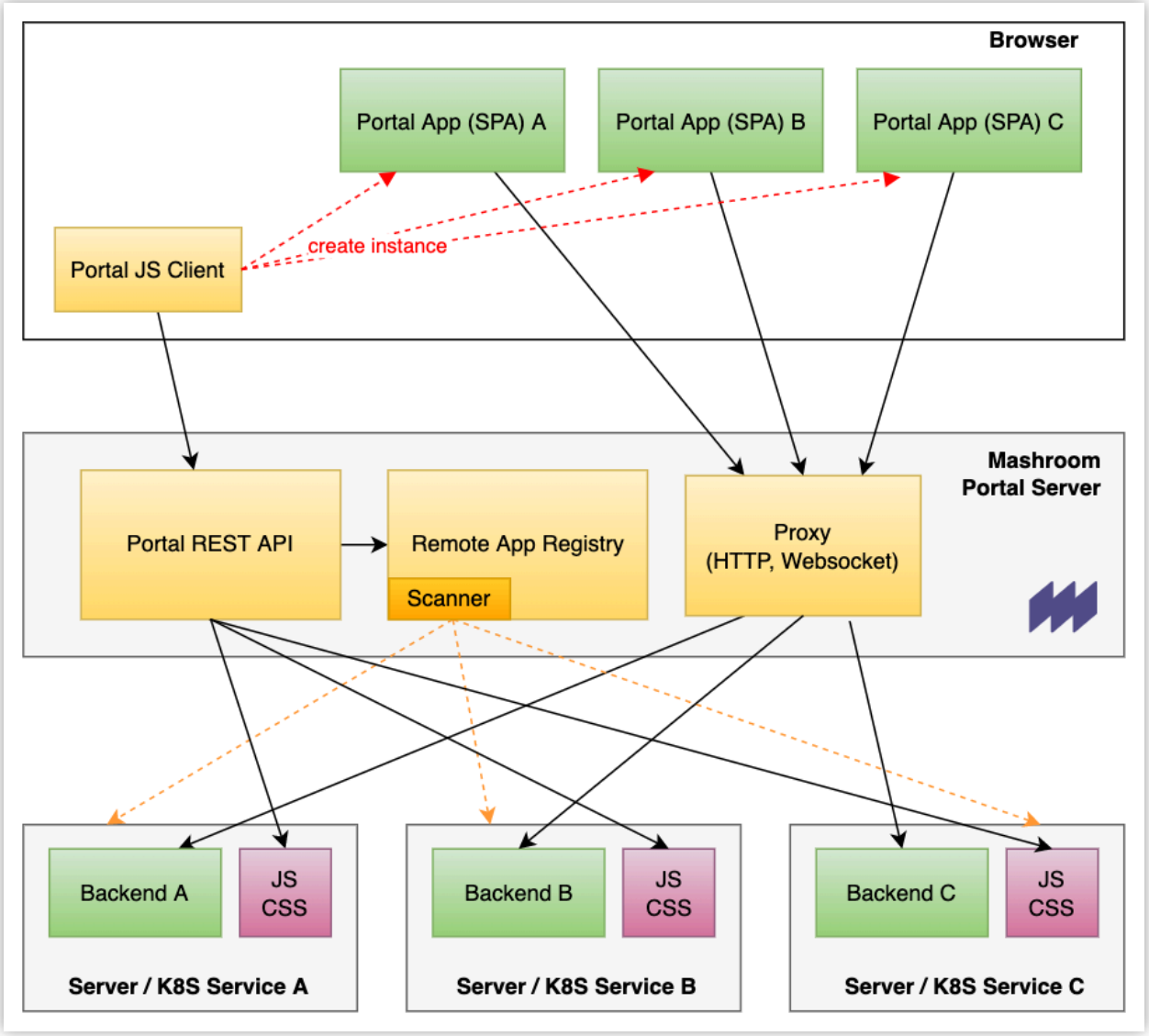
Checkout out the mashroom-http-proxy documentation for details.

Remote Apps

Portal Apps (SPA) can reside on a remote server and automatically be registered. Currently, there are two different remote registries that can also be combined:

- mashroom-portal-remote-app-registry
- mashroom-portal-remote-app-registry-k8s (for Kubernetes)

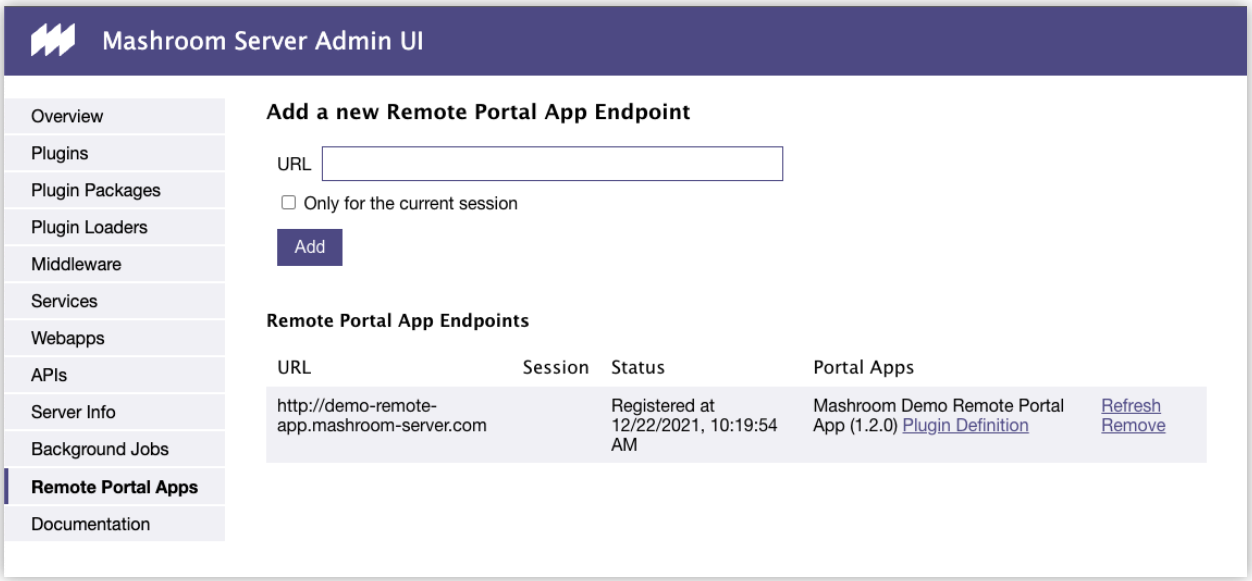
Here, how it works:



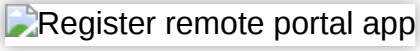
Ad hoc register a remote app

Here for example with mashroom-portal-remote-app-registry.

Open */mashroom/admin/ext/remote-portal-apps*, paste the URL into the input and lick *Add*:



After that you can add the new Portal App via Drag'n'Drop where ever you want:



Server Side Rendering

Since v2 the Portal can render the whole Portal page on the server-side and even include the initial HTML of Portal Apps, capable of server-side rendering.

> **NOTE**: If an App support SSR it should provide its style in separate CSS file, because in that case the Portal will try to inline all styles so everything gets rendered correctly without any extra resources.

To enable SSR in an App you have to:

- Use the new plugin type *portal-app2*
- Add a *ssrBootstrap* that will be executed on the server side
- Check in the (client-side) *bootstrap* if there is a pre-rendered DOM and use *hydrate()* in that case

The *ssrBootstrap* could look like this in React:

```
import React from 'react';
import {renderToString} from 'react-dom/server';
import App from './App';

import type {MashroomPortalAppPluginSSRBootstrapFunction} from '@mashroom/mashroom-p

const bootstrap: MashroomPortalAppPluginSSRBootstrapFunction = (portalAppSetup) => {
    const {appConfig, restProxyPaths, lang} = portalAppSetup;
    const dummyMessageBus: any = {};
    return renderToString(<App appConfig={appConfig} messageBus={dummyMessageBus}/>)
};

export default bootstrap;
```

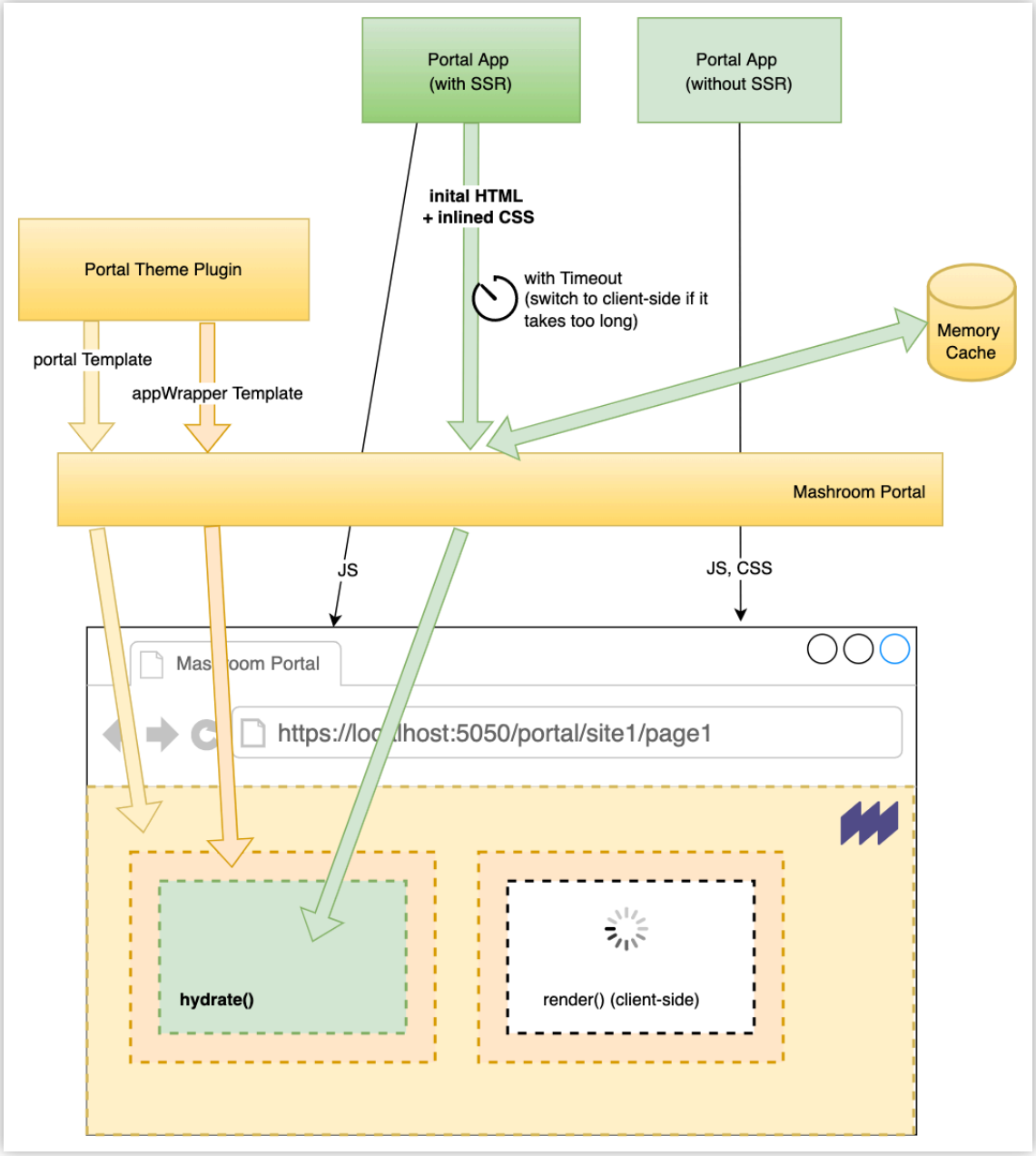
> **NOTE**: The server-side bootstrap will receive the same *portalAppSetup* like on the client-side, but no client services. If you need the *messageBus* or other services make sure the App renders without them or with a mock implementation.

On the client-side you would do:

```
import React from 'react';
import {render, hydrate, unmountComponentAtNode} from 'react-dom';
import App from './App';

import type {MashroomPortalAppPluginBootstrapFunction} from '@mashroom/mashroom-port

const bootstrap: MashroomPortalAppPluginBootstrapFunction = (element, portalAppSetup
    const {appConfig, restProxyPaths, lang} = portalAppSetup;
    const {messageBus} = clientServices;

    const ssrHost = element.querySelector('[data-ssr-host="true"]');
    if (ssrHost) {
        hydrate(<App appConfig={appConfig} messageBus={messageBus}/>, ssrHost);
    } else {
        render(<App appConfig={appConfig} messageBus={messageBus}/>, element);
    }
};

global.startMyApp = bootstrap;
```

Remote Apps would basically work the same, but the server-side bootstrap needs to be exposed as route that receives a POST with the *portalAppSetup* in the content. Checkout the Mashroom Demo SSR Portal App .

All put together rendering a page works like this:



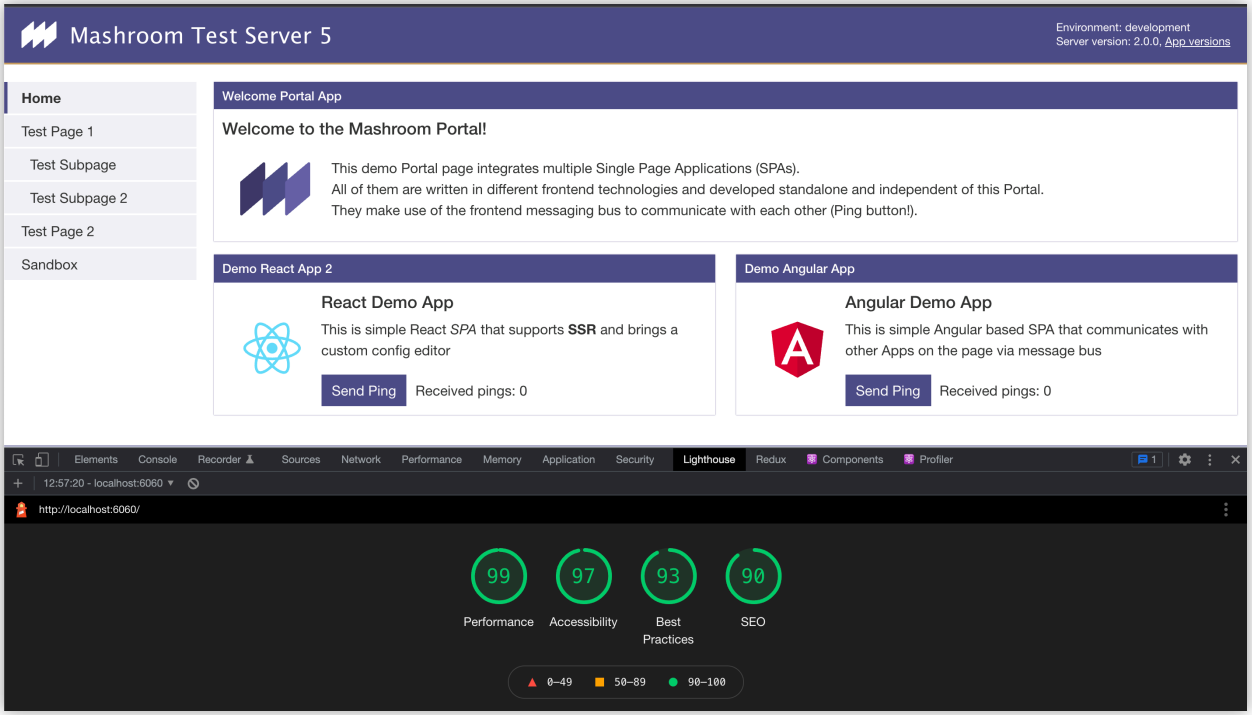
NOTE: Every App should support client-side rendering, because they can be added dynamically to a page. If you don't want to render on the client (e.g. for security reasons) at least render an error message on the client-side.

Performance/SEO hints

SSR improves page performance and SEO heavily. To improve it further we recommend:

- Consider the *renderTimoutMs* value in the Portal's *ssrConfig*. This timeout determines how long the Portal will wait for an SSR result. It is safe to use a very low value, because if the SSR takes to long it automatically switches to client-side rendering, but puts the result (when ready) into the memory cache. So, subsequent calls will get the cached SSR HTML.

- Use the mashroom-memory-cache, because the Portal will use it to cache the results of server-side rendering
- Use a reverse proxy for TLS termination and enable compression and caching there
- Use a CDN to accelerate the delivering of resources
- In your server-side rendering Apps make sure to avoid Cumulative Layout Shift (e.g. by always defining height and width for media)
- Use the Google Lighthouse to check your site:



"SPA Mode"

The Portal supports dynamically replacing the content of a page (that's the layout + all Apps) by the content of another page. This can be used to avoid full page loads and let the Portal behave like an SPA itself, so it basically can switch to client-side rendering.

The client-side rendering needs to be implemented in the *Theme*, which needs to the following during page navigation:

- Update the URL via *window.history*
- Update the navigation area (highlight the new page)
- Fetch the page content of the target page:

```
<!-- in the template -->
<a class="nav-link active"
   href="{{@root.siteBasePath}}{{friendlyUrl}}"
   onclick="return replacePageContent('{{pageId}}', '{{@root.siteBasePath}}{{friendl
   data-mr-page-id="{{pageId}}">
    {{title}}
</a>
```

```
import type {MashroomPortalClientServices, MashroomPortalPageContent} from '@mashroc

const clientServices: MashroomPortalClientServices | undefined = (global as any).Mas
if (!clientServices) {
    return;
}

(global as any).replacePageContent = (pageId: string, pageUrl: string): boolean => {
    showPageLoadingIndicator(true);
    clientServices.portalPageService.getPageContent(pageId).then(
        (content: MashroomPortalPageContent) => {
            if (content.fullPageLoadRequired || !content.pageContent) {
                // Full page load required!
                document.location.replace(pageUrl);
            } else {
                contentEl.innerHTML = content.pageContent;
                // Execute page scripts
                eval(content.evalScript);
                highlightPageIdInNavigation(pageId);
                window.history.pushState({ pageId }, '', pageUrl);
                showPageLoadingIndicator(false);
            }
        },
        (error) => {
            // If an error occurs we do a full page load
            console.error('Dynamically replacing the page content failed!', error);
            document.location.replace(pageUrl);
        }
    );
    return false;
}
```

NOTE: The Portal will automatically detect if the requested page is not compatible to the initial loaded one (because the Theme oder Page Enhancements differ). In that case it will return `fullPageLoadRequired: true`.

Composite Apps

A *Composite App* is a higher order App, which uses other Portal Apps (SPAs) as building blocks. So, its App in App, or SPA in SPA, or picture in picture ;-) Such a Composite App can itself be used as building block for another Composite App, which can be continued infinite.

This approach takes advantage of *Mashroom Portal*'s ability to load any registered App into any DOM element. Basically, you just render a *div* element in your SPA with a unique (random) ID and load the App like so:

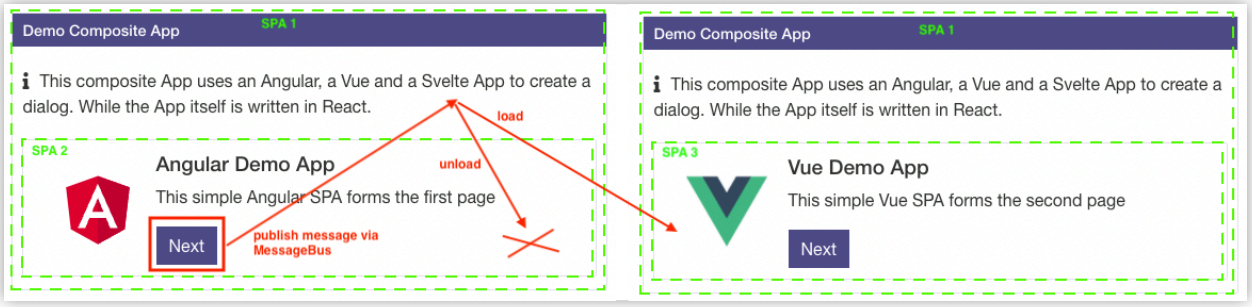
```
// Get the portalAppService in the bootstrap
const bootstrap: MashroomPortalAppPluginBootstrapFunction = (element, portalAppSetup
    const {portalAppService} = clientServices;
    //...
}

// Load the App
const loadedApp = await portalAppService.loadApp(domElID, 'My App', null, /* positic
    someProp: 'foo',
});
```

And unload it like this:

```
await portalAppService.unloadApp(loadedApp.id);
```

Make sure you call unloadApp() if you want to remove/replace an App and not just remove the host DOM node, because like that the resources are not properly removed from the browser.

Here for example the mashroom-portal-demo-composite-app:



NOTE: A Composite App only runs in *Mashroom Portal* and leads to a vendor lock-in. At very least other integration hosts need to provide a compatible implementation of *MashroomPortalAppService*.

Dynamic Cockpits

One key feature of *Mashroom Portal* is the possibility to create dynamic pages (or cockpits) based:

- Backend data (e.g. a search)

- The available (registered) Portal Apps
- Messages published on the message bus

This is quite similar to a *Composite App* but in this case the page loads a single (static) App, which takes over the page and manages loading and unloading other Apps *outside* of itself. Typically, the names of the Apps that get loaded are not pre-defined, but determined from some *metaInfo* of the available Apps. So, such a cockpit can be dynamically extended by just adding new Apps with some *capabilities* (at runtime).

To find Apps you would use the *portalAppService* like this:

```
const availableApps = await portalAppService.getAvailableApps();
const appToShowWhatever = availableApps.filter(({metaInfo}) => metaInfo?.canShow ===
```

Have a look at this Demo Dynamic Cockpit , it consist of a central search bar and tries to load found data with Apps with apropriate *metaInfo*. E.g. this App could load customers:

```
{
    "plugins": [
        {
            "name": "Mashroom Dynamic Cockpit Demo Customer Details App",
            // ...
            "defaultConfig": {
                "metaInfo": {
                    "demoCockpit": {
                        "viewType": "Details",
                        "entity": "Customer"
                    }
                },
                // ...
            }
        }
    ]
}
```

Theming

A *Mashroom Portal* Theme can be written with any template engine that works with Express .

You need to implement three templates:

- *portal*: A Portal page
- *appWrapper*: Renders the Portal App wrapper (optional, can be omitted)

- *appError*: Renders the error if loading an App fails (optional, can be omitted)

> **NOTE**: If you want to have a type-safe template we recommend using React . Have a look at mashroom-portal-demo-alternative-theme for an example.

Themable Portal Apps

Obviously, *theming* only works properly if all the Apps on a page regard the currently applied theme and in particular **do not**:

- Just hardcode colors
- Override "global" styles (e.g. for headers, inputs, etc.)
- Bring their own fonts

Here some best practices:

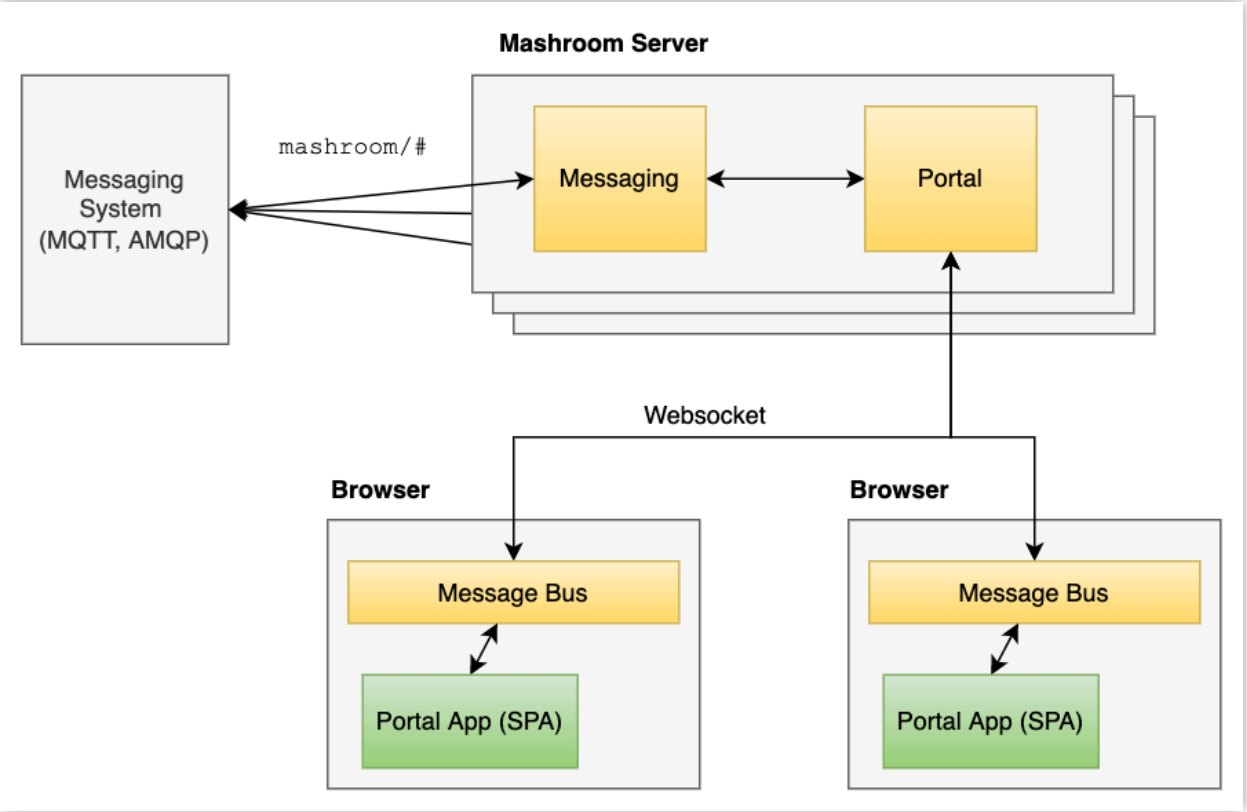
- In your Theme
  - Ship the style for all HTML elements and a common grid system (such as bootstrap)
  - Define your UI component library in pure CSS without any JavaScript or markup (like bootstrap does it)
  - Expose your design constants (colors, fonts, …) via CSS variables or utility classes
- In your Apps:
  - Implement your UI components with the framework of your choice but use the CSS classes and the given structure from the Theme
  - Use only custom classes to apply your style within the App
  - Make sure your custom classes don't collide by using CSS modules or prefixing them with some unique key
  - For colors, fonts and such use the CSS variables or utility classes exposed by the Theme
  - To be able to run your Apps standalone make sure the relevant parts of the Theme are available as standalone CSS file

Messaging

The Portal comes with a client-side *MessageBus* that can be used by Portal Apps to communicate with each other locally.

If server-side messaging (*mashroom-messaging*) and Websocket support (*mashroom-websocket*) is installed, the *MessageBus* is automatically connected to the server-side

messaging facility and like this Portal Apps can communicate with Apps in other browsers and even with 3rd party systems (when a external messaging system such as MQTT is connected).



Page Enhancements

Page enhancement plugins allow to add some extra script or style to a page. Either to any page or based on some rules (e.g. page URL or the user agent).

Checkout the mashroom-portal documentation for details.

Portal App Enhancements

Portal app enhancement plugins can be used to modify the *portalAppSetup* of some (or any) Apps before loading. It can also be used to add custom services on the client side (passed with *clientServices* to every App).

A typical use case would be to add some extra data to the *user* or to augment the *appConfig*.

Checkout the mashroom-portal documentation for details.

Security

The Mashroom Portal uses the security plugin to control the access to pages and Portal Apps. It also introduced a concept of fine grain permissions (mapped to roles) which can be checked in Portal Apps and in backends (passed via HTTP header by the API Proxy).

Portal App Security

If a user requires specific roles to be able to load an App (dynamically) you have to set the *defaultRestrictViewToRoles* in the plugin definition. Otherwise, any user, which is able to access the Portal API (which can be controlled via ACL), will be able to load it.

NOTE: Users with the role *Administrator* are able to load **all** Apps, regardless of *defaultRestrictViewToRoles*. They are also able to override/redefine the *view* permission when adding Apps to a page.

Within Portal Apps you should not work with roles but with abstract *permission* keys such as "mayDeleteCustomer". In the plugin definition this keys can then be mapped to roles like this:

```
"defaultConfig": {
  "rolePermissions": {
    "mayDeleteCustomer": ["Role1", "Role2"]
  }
}
```

And the *permission* can be checked like this:

```
const bootstrap: MashroomPortalAppPluginBootstrapFunction = (portalAppHostElement, p
    const {appConfig, user: {permissions, username, displayName, email}} = portalApp

    // True if user has role "Role1" OR "Role2"
    if (permissions.mayDeleteCustomer) {
        // ...
    }
}
```

Securing backend access

Proxy access can be restricted by adding a *restrictToRole* property in the plugin definition:

```
"defaultConfig": {
  "proxies": {
    "myApi": {
      "targetUri": "http://localhost:1234/api/v1",
      "sendPermissionsHeader": false,
      "restrictToRoles": ["Role1"]
    }
  }
}
```

NOTE: Not even users with the *Administrator* role can override that restriction. So, even if they can load a restricted App they will not be able to access the backend.

Furthermore, it is possible to pass the Portal App security context to the backend via HTTP headers. This is useful if you want to check some fine grain permissions there as well.

Headers that can be passed to the Backend:

- X-USER-PERMISSIONS: A comma separated list of the permission keys that resolve to true; can be activated by setting *sendPermissionsHeader* to true on the proxy definition
- X-USER-NAME and others: Can be activated by adding the http-proxy-add-user-headers plugin
- X-USER-ACCESS-TOKEN: The access token if the OpenID-Connect is used a security provider; can be activated by adding the mashroom-http-proxy-add-access-token plugin

Site and Page Security

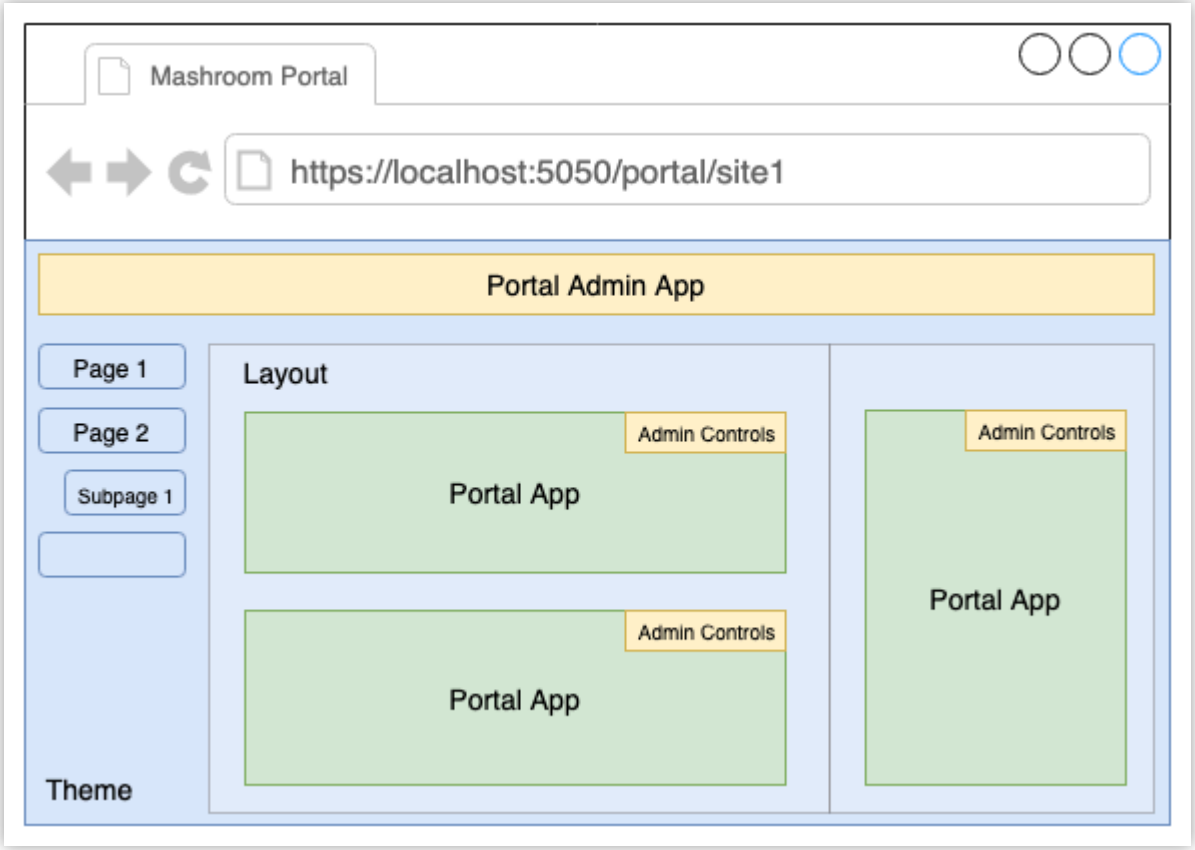
Sites and Pages can be secured by:

- The ACL
- Individually, via Admin Toolbar (see below)

Both approaches can be combined.

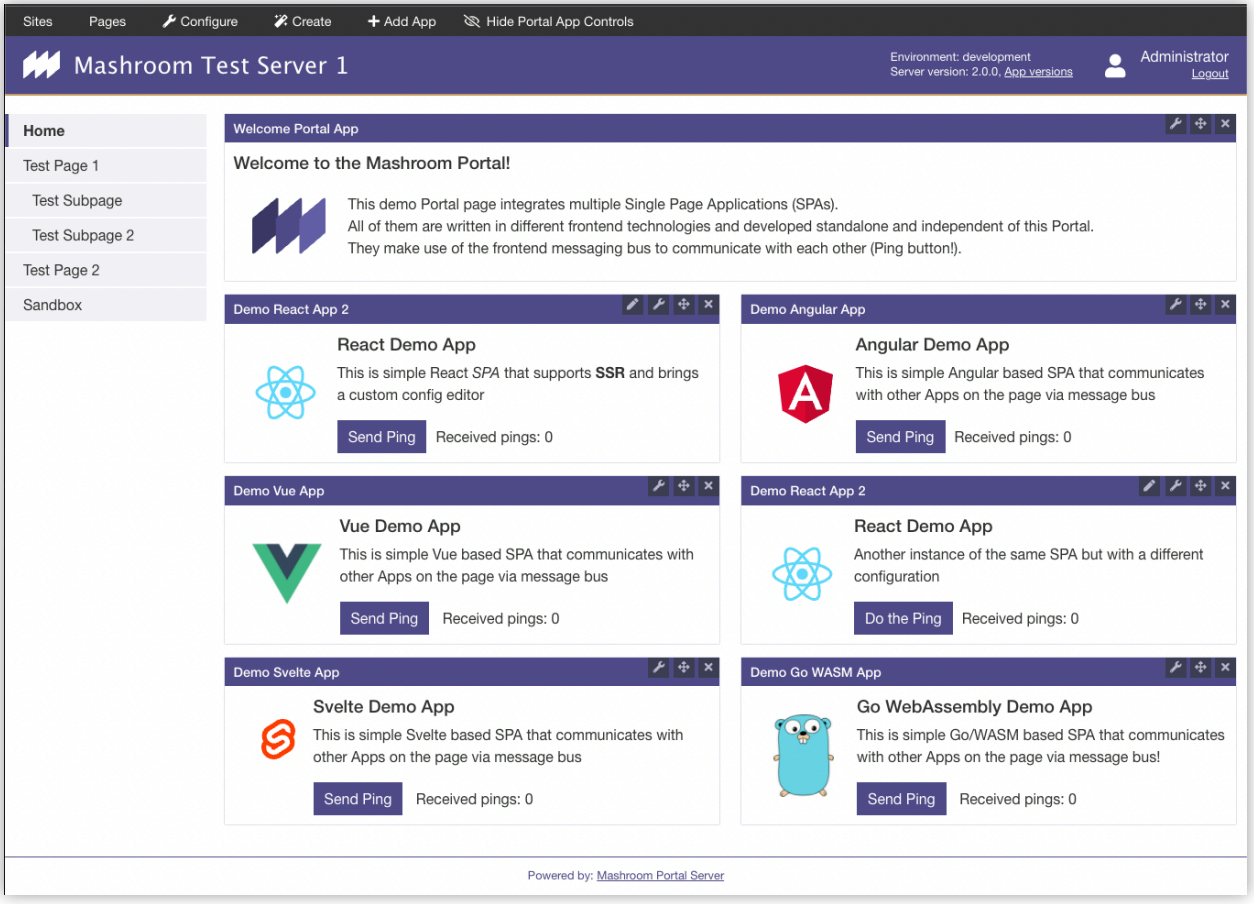
User Interface

Page layout



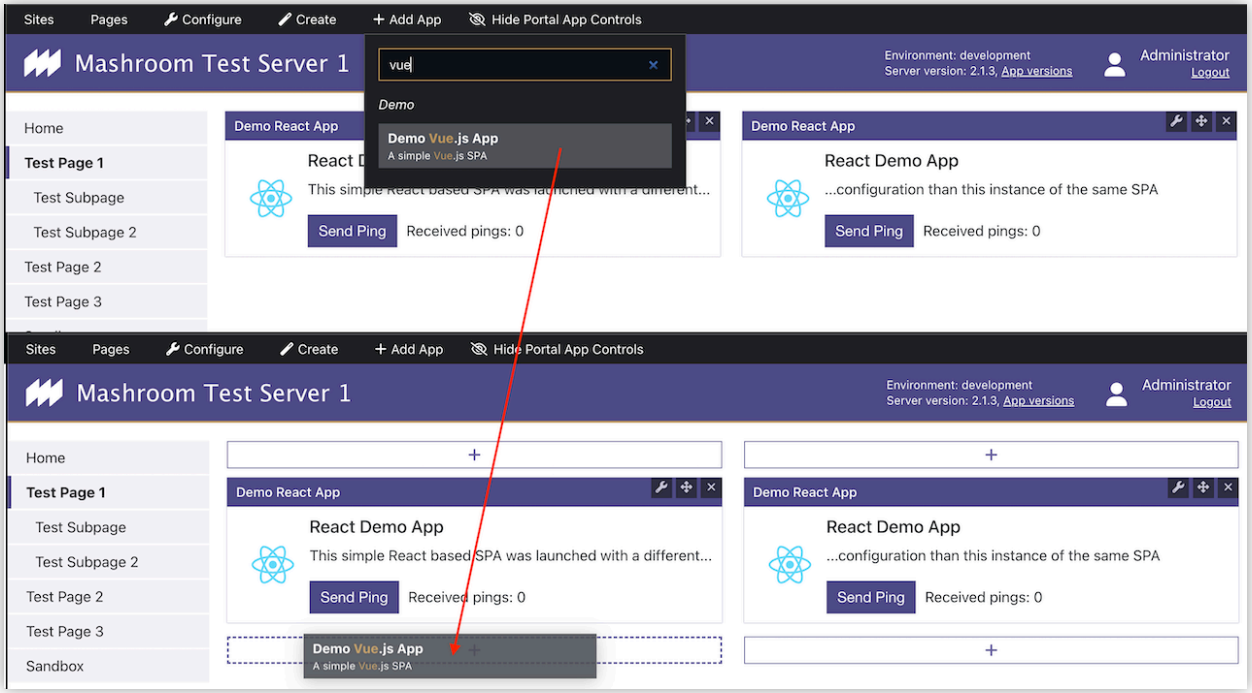
The *theme* is responsible for rendering the page. It defines where the main content is. The portal adds the selected layout to the main content and the configured Portal Apps within the app areas of this layout.

Default start page with demo apps



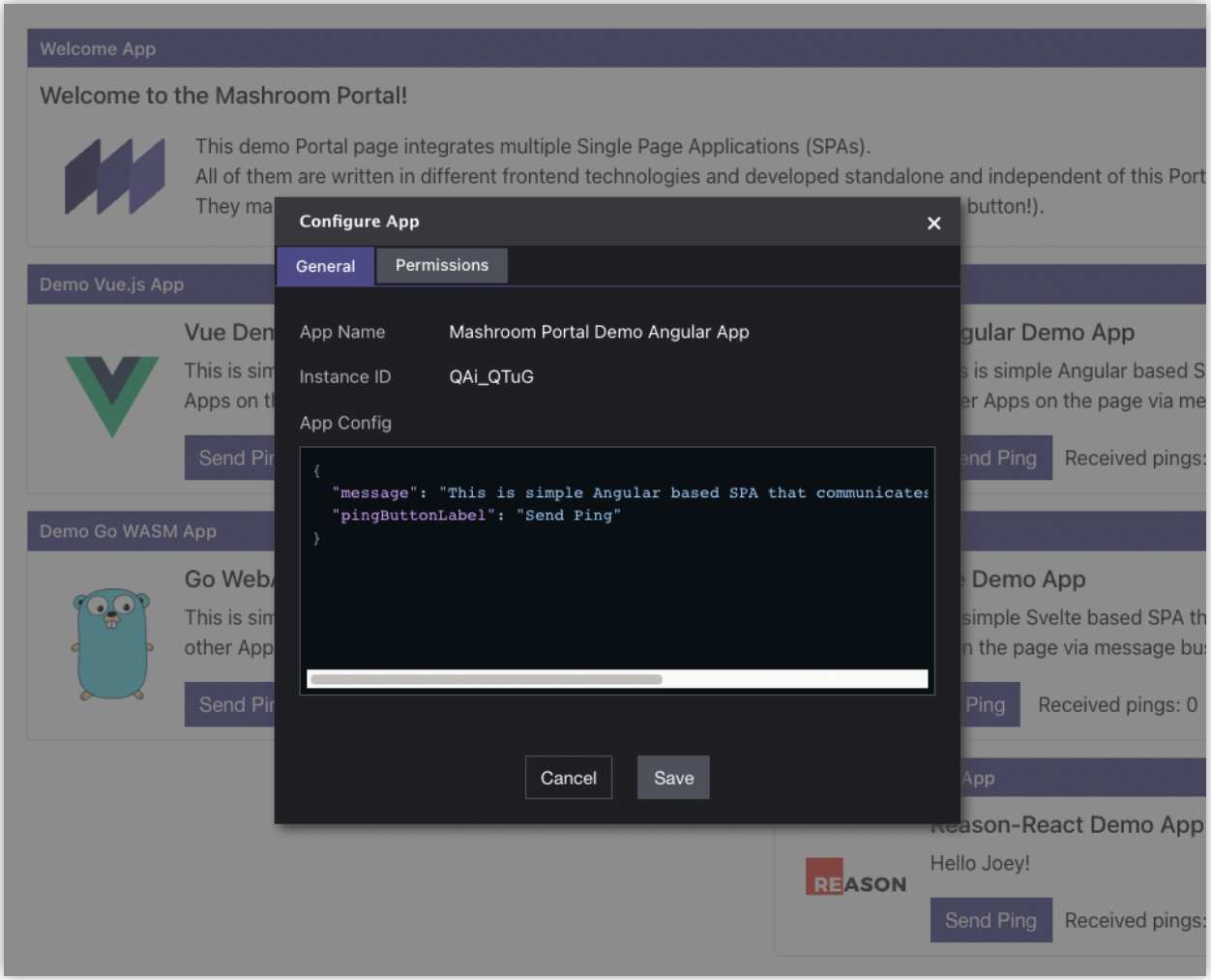
Add a new Portal App

As an *Administrator* you can add Portal Apps via Admin Toolbar: *Add Apps*



Portal App configuration

After adding an App you can click on the *Configure* icon to edit the *appConfig* and the permissions:

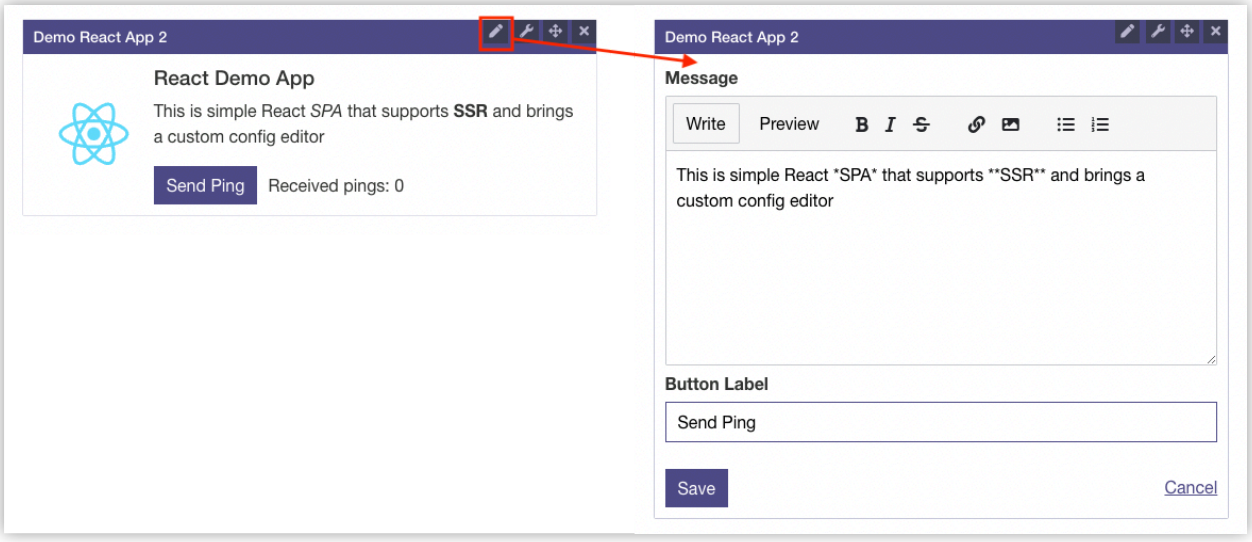


Custom App Config Editor

Instead of the default JSON editor you can define a custom editor App for you *appConfig*. The custom editor is itself a plain Portal App (SPA) which gets an extra *appConfig* property *editorTarget* of type *MashroomPortalConfigEditorTarget* that can be used to communicate back with the Admin Toolbar:

```
const bootstrap: MashroomPortalAppPluginBootstrapFunction = (portalAppHostElement, p
    const {appConfig: {editorTarget}} = portalAppSetup;

    if (!editorTarget || !editorTarget.pluginName) {
        throw new Error('This app can only be started as App Config Editor!');
    }

    const currentAppConfig = editorTarget.appConfig;

    // ...

    // When the user is done:
    editorTarget.updateAppConfig(updatedAppConfig);
    editorTarget.close();

    //...
};
```

Here for example the mashroom-portal-demo-react-app2 plugin which has a custom editor:

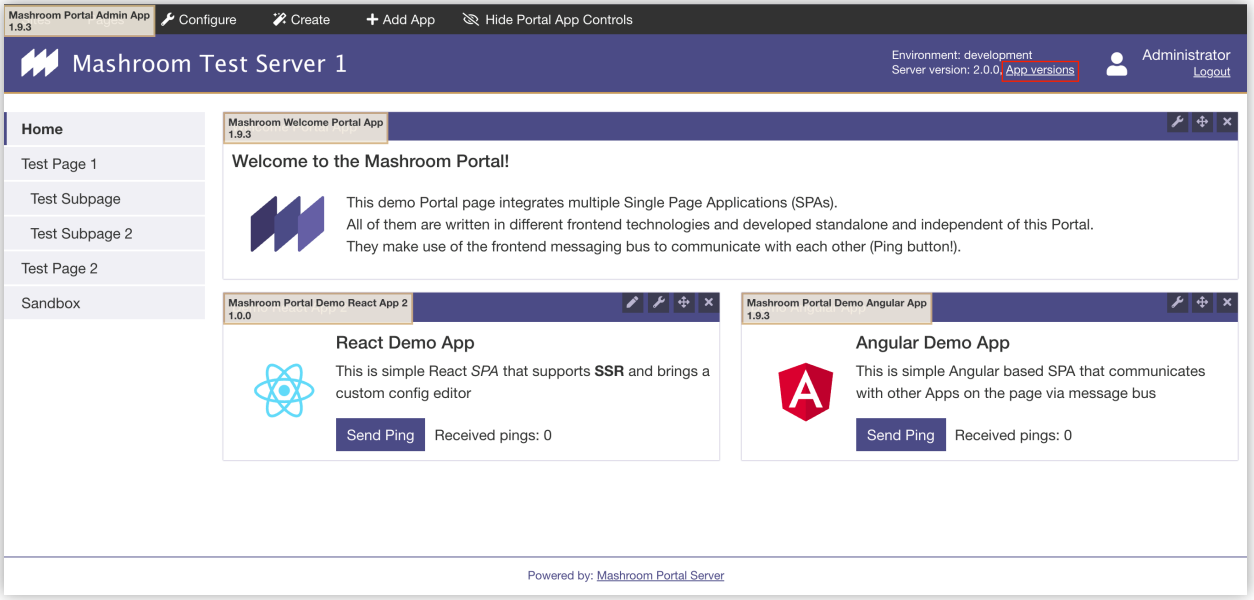


Show Portal App versions

The default portal theme allows to show all Portal App versions by clicking on *App versions*. You can enable it like this in the *Mashroom* config file:

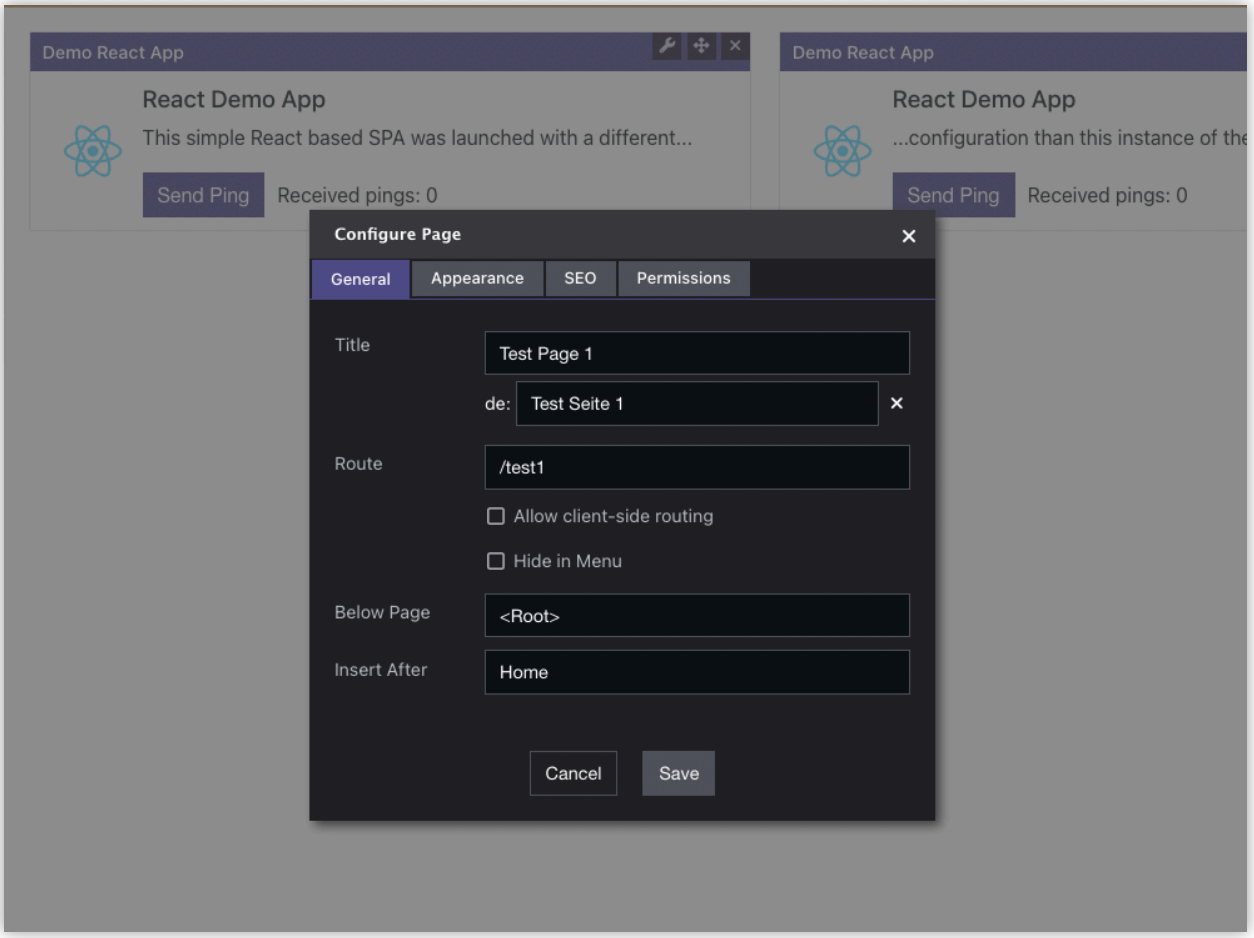
```
{
    "plugins": {
        "Mashroom Portal Default Theme": {
          "showEnvAndVersions": true
```

```
        }
    }
}
```



Adding a new page

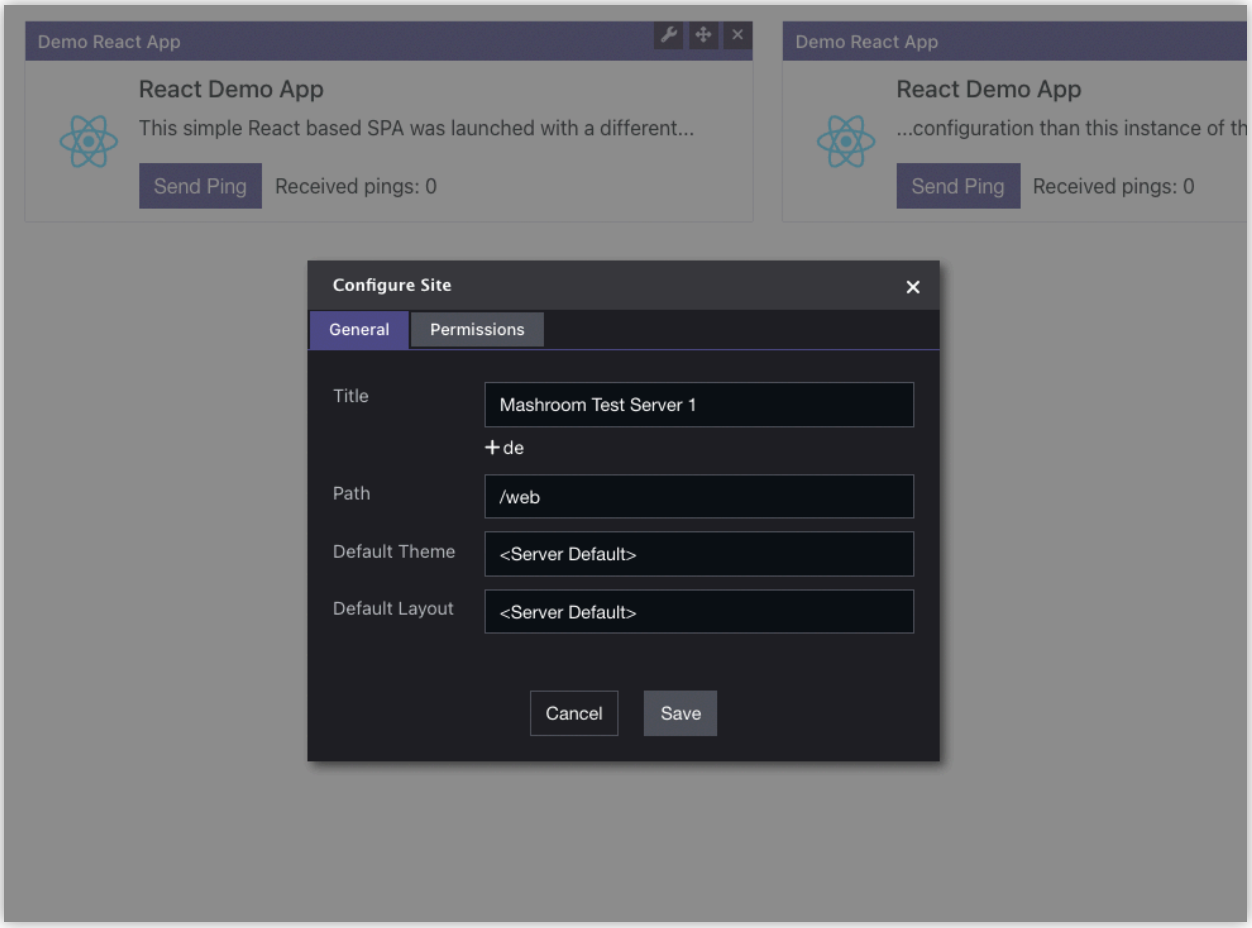
As an *Administrator* you can add a new Page from the Admin Toolbar: *Create -> Create New Page*:



After that you can start to place Portal Apps via *Add Apps*.

Adding a new site

As an *Administrator* you can add a new Site from the Admin Toolbar: *Create -> Create New Page*:



After that you can start to add additional pages.

Core Documentation

Services

MashroomPluginService

Accessible through *pluginContext.services.core.pluginService*

Interface:

```
export interface MashroomPluginService {
    /**
     * The currently known plugin loaders
     */
    getPluginLoaders(): Readonly<MashroomPluginLoaderMap>;

    /**
     * Get all currently known plugins
     */
    getPlugins(): Readonly<Array<MashroomPlugin>>;

    /**
     * Get all currently known plugin packages
     */
    getPluginPackages(): Readonly<Array<MashroomPluginPackage>>;

    /**
     * Register for the next loaded event of given plugin (fired AFTER the plugin ha
     */
    onLoadedOnce(pluginName: string, listener: () => void): void;

    /**
     * Register for the next unload event of given plugin (fired BEFORE the plugin i
     */
    onUnloadOnce(pluginName: string, listener: () => void): void;
}
```

MashroomMiddlewareStackService

Accessible through *pluginContext.services.core.middlewareStackService*

Interface:

```
export interface MashroomMiddlewareStackService {
    /**
     * Check if the stack has given plugin
     */
    has(pluginName: string): boolean;

    /**
     * Execute the given middleware.
     * Throws an exception if it doesn't exists
     */
    apply(
        pluginName: string,
        req: Request,
        res: Response,
    ): Promise<void>;

    /**
     * Get the ordered list of middleware plugin (first in the list is executed firs
     */
    getStack(): Array<{pluginName: string; order: number}>;
}
```

MashroomHttpUpgradeService

Accessible through *pluginContext.services.core.websocketUpgradeService*

Interface:

```
/**
 * A services to add and remove HTTP/1 upgrade listeners
 */
export interface MashroomHttpUpgradeService {
    /**
     * Register an upgrade handler for given path
     */
    registerUpgradeHandler(handler: MashroomHttpUpgradeHandler, pathExpression: stri
    /**
     * Unregister an upgrade handler
     */
    unregisterUpgradeHandler(handler: MashroomHttpUpgradeHandler): void;
}
```

MashroomHealthProbeService

A service that allows it plugins to register health probes. If a probe fails the server state goes to unready.

```
/**
 * A services to obtain all available health probes
 */
export interface MashroomHealthProbeService {
    /**
     * Register a new health probe for given plugin
     */
    registerProbe(forPlugin: string, probe: MashroomHealthProbe): void;
    /**
     * Unregister a health probe for given plugin
     */
    unregisterProbe(forPlugin: string): void;
    /**
     * Get all registered probes
     */
    getProbes(): Readonly<Array<MashroomHealthProbe>>;
}
```

You can use it like this in your plugin bootstrap:

```
const bootstrap: MashroomStoragePluginBootstrapFunction = async (pluginName, pluginC
    const {services: {core: {pluginService, healthProbeService}}} = pluginContextHol

    healthProbeService.registerProbe(pluginName, healthProbe);

    pluginService.onUnloadOnce(pluginName, () => {
        healthProbeService.unregisterProbe(pluginName);
    });

    // ...
};
```

Plugin Types

plugin-loader

A *plugin-loader* plugin adds support for a custom plugin type.

To register a new plugin-loader add this to *package.json*:

```
{
    "mashroom": {
        "plugins": [
            {
                "name": "My Custom Plugin Loader",
                "type": "plugin-loader",
                "bootstrap": "./dist/mashroom-bootstrap",
                "loads": "my-custom-type",
```

```
                "defaultConfig": {
                   "myProperty": "foo"
                }
            }
        ]
    }
}
```

- *loads*: The plugin type this loader can handle

After that all plugins of type *my-custom-type* will be passed to your custom loader instantiated by the bootstrap script:

```
import type {
    MashroomPluginLoader, MashroomPlugin, MashroomPluginConfig, MashroomPluginContex
    MashroomPluginLoaderPluginBootstrapFunction
} from 'mashroom/type-definitions';

class MyPluginLoader implements MashroomPluginLoader {

    get name(): string {
        return 'My Plugin Loader';
    }

    generateMinimumConfig(plugin: MashroomPlugin) {
        return {};
    }

    async load(plugin: MashroomPlugin, config: MashroomPluginConfig, context: Mashrc
        // TODO
    }

    async unload(plugin: MashroomPlugin) {
        // TODO
    }
}

const myPluginLoaderPlugin: MashroomPluginLoaderPluginBootstrapFunction = (pluginNam
    return new MyPluginLoader();
};

export default myPluginLoaderPlugin;
```

web-app

Registers a *Express* webapp that will be available at a given path.

To register a web-app plugin add this to *package.json*:

```
{
     "mashroom": {
        "plugins": [
            {
                "name": "My Webapp",
                "type": "web-app",
                "bootstrap": "./dist/mashroom-bootstrap.js",
                "defaultConfig": {
                    "path": "/my/webapp",
                    "myProperty": "foo"
                }
            }
        ]
     }
}
```

- *defaultConfig.path*: The default path where the webapp will be available

And the bootstrap just returns the *Express* webapp:

```
import webapp from './webapp';

import type {MashroomWebAppPluginBootstrapFunction} from '@mashroom/mashroom/type-de

const bootstrap: MashroomWebAppPluginBootstrapFunction = async () => {
    return webapp;
};

export default bootstrap;
```

Additional handlers

It is also possible to return handlers in the bootstrap. Currently there is only one:

- *upgradeHandler*: Handle HTTP Upgrades (e.g. upgrade to WebSocket). Alternatively you could use *MashroomWebsocketUpgradeService* directly

Example:

```
const bootstrap: MashroomWebAppPluginBootstrapFunction = async () => {
    return {
        expressApp: webapp,
        upgradeHandler: (request: IncomingMessageWithContext, socket: Socket, head:
            // TODO
        },
```

```
    };
};
```

api

Registers a *Express Router* (a REST API) and makes it available at a given path.

To register a API plugin add this to *package.json*:

```
{
     "mashroom": {
        "plugins": [
            {
                "name": "My REST API",
                "type": "api",
                "bootstrap": "./dist/mashroom-bootstrap.js",
                "defaultConfig": {
                    "path": "/my/api",
                    "myProperty": "foo"
                }
            }
        ]
    }
}
```

- *defaultConfig.path*: The default path where the api will be available

And the bootstrap just returns the *Express* router:

```
const express = require('express');
const router = express.Router();

router.get('/', (req, res) => {
  // ...
});

import type {MashroomApiPluginBootstrapFunction} from '@mashroom/mashroom/type-defir

const bootstrap: MashroomApiPluginBootstrapFunction = async () => {
    return router;
};

export default bootstrap;
```

middleware

Registers a *Express* middleware and adds it to the global middleware stack.

To register a middleware plugin add this to package.json:

```
{
    "mashroom": {
        "plugins": [{
            "name": "My Middleware",
            "type": "middleware",
            "bootstrap": "./dist/mashroom-bootstrap.js",
            "defaultConfig": {
                "order": 500,
                "myProperty": "foo"
            }
        }]
    }
}
```

- *defaultConfig.order*: The weight of the middleware in the stack - the higher it is the **later** it will be executed (Default: 1000)

And the bootstrap just returns the *Express* middleware:

```
import MyMiddleware from './MyMiddleware';

import type {MashroomMiddlewarePluginBootstrapFunction} from '@mashroom/mashroom/typ

const bootstrap: MashroomMiddlewarePluginBootstrapFunction = async (pluginName, plug
    const pluginContext = pluginContextHolder.getPluginContext();
    const middleware = new MyMiddleware(pluginConfig.myProperty, pluginContext.logge
    return middleware.middleware();
};

export default bootstrap;
```

static

Registers some static resources and exposes it at a given path (via *Express* static).

To register a static plugin add this to package.json:

```
{
    "mashroom": {
        "plugins": [{
            "name": "My Documents",
            "type": "static",
            "documentRoot": "./my-documents",
            "defaultConfig": {
                "path": "/my/docs"
```

```
            }
        }]
    }
}
```

- *documentRoot*: Defines the local root path of the documents
- *defaultConfig.path*: The default path where the documents will be available

services

Used to load arbitrary shared code that can be loaded via *pluginContext*.

To register a service plugin add this to package.json:

```
{
    "mashroom": {
        "plugins": [{
            "name": "My Services",
            "type": "services",
            "namespace": "myNamespace",
            "bootstrap": "./dist/mashroom-bootstrap.js",
            "defaultConfig": {
            }
        }]
    }
}
```

- *namespace*: Defines the path to the services. In this case *MyService* will be accessible through *pluginContext.services.myNamespace.service*

The bootstrap will just return an object with a bunch of services:

```
import MyService from './MyService';

import type {MashroomServicesPluginBootstrapFunction} from '@mashroom/mashroom/type-

const bootstrap: MashroomServicesPluginBootstrapFunction = async (pluginName, plugir
    const pluginContext = pluginContextHolder.getPluginContext();
    const service = new MyService(pluginContext.loggerFactory);

    return {
        service,
    };
};
```

```
export default bootstrap;
```

admin-ui-integration

A simple plugin to register an arbitrary *web-app* or *static* plugin as panel in the Admin UI.

To register an admin-ui-integration plugin add this to package.json:

```
{
    "mashroom": {
        "plugins": [{
            "name": "My Admin Panel Integration",
            "type": "admin-ui-integration",
            "requires": [
                "My Admin Panel"
            ],
            "target": "My Admin Panel",
            "defaultConfig": {
                "menuTitle": "My Admin Panel",
                "path": "/my-admin-panel",
                "height": "80vh",
                "weight": 10000
            }
        }]
    }
}
```

- *target*: The actual web-app or static plugin that should be integrated
- *defaultConfig.menuTitle*: The name that should be appear in the Admin UI menu
- *defaultConfig.path*: The path in the Admin UI (full path will be /mashroom/admin/ext/\)
- *defaultConfig.height*: The height of the iframe that will contain the target webapp (Default: 80vh) If you want that the iframe has the full height of the webapp you have to post the height periodically to the parent, like so

```
    parent.postMessage({ height: contentHeight + 20 }, "*");
```

- *defaultConfig.weight*: The weight of the menu entry, the higher the number the lower will be menu entry be (Default: 100)

Plugin Documentation

Mashroom Security

This plugin adds role based security to the *Mashroom Server*.

It comes with the following mechanisms:

- A new *Security Provider* plugin type that does the actual authentication and can be used to obtain the user roles
- An access control list (ACL) based on a JSON file that can be used to protect paths and HTTP methods based on roles or IP addresses
- A middleware that checks for every request the ACL and if authentication and specific roles are required. If authentication is required and no user present it triggers an authentication (via *Security Provider*).
- A shared service to programmatically restrict the access to resources such as Pages or Apps (used by the *Mashroom Portal*)

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-security** as *dependency*.

You can override the default config in your Mashroom config file like this:

```
{
  "plugins": {
        "Mashroom Security Services": {
            "provider": "Mashroom Security Simple Provider",
            "forwardQueryHintsToProvider": [],
            "acl": "./acl.json"
        }
    }
}
```

- *provider*: The plugin that actually does the authentication and knows how to retrieve the user roles (Default: Mashroom Security Simple Provider)

- *forwardQueryHintsToProvider*: A list of query parameters that should be forwarded during the authentication. (will be added to the login or authorization URL).

- *acl*: The ACL for path based security restrictions (see below) (Default: ./acl.json)

ACL

A typical ACL configuration looks like this:

```
{
    "$schema": "https://www.mashroom-server.com/schemas/mashroom-security-acl.json",
    "/portal/**": {
        "*": {
            "allow": {
                "roles": ["Authenticated"]
            }
        }
    },
    "/mashroom/**": {
        "*": {
            "allow": {
                "roles": ["Administrator"],
                "ips": ["127.0.0.1", "::1"]
            }
        }
    }
}
```

The general structure is:

```
    "/my/path/**": {
        "*|GET|POST|PUT|DELETE|PATCH|OPTIONS": {
            "allow": "any"|<array of roles>|<object with optional properties roles a
            "deny": "any"|<array of roles>|<object with optional properties roles ar
        }
    }
```

- The path can contain the wildcard "*" for single segments and "**" for multiple segments

- "any" includes anonymous users

- IP addresses can also contain wildcards: "?" for s single digit, "*" for single segments and "**" for multiple segments

Example: Allow all users except the ones that come from an IP address starting with 12:

```
{
    "/my-app/**": {
        "*": {
            "allow": {
                "roles": ["Authenticated"]
            },
            "deny": {
                "ips": ["12.**"]
            }
        }
    }
}
```

Example: Restrict the Portal to authenticated users but make a specific site public:

```
{
    "/portal/public-site/**": {
        "*": {
            "allow": "any"
        }
    },
    "/portal/**": {
        "*": {
            "allow": {
                "roles": ["Authenticated"]
            }
        }
    }
}
```

Security Service

Adding and checking a resource permission (e.g. for a Page) works like this:

```
import type {MashroomSecurityService} from '@mashroom/mashroom-security/type-definit

export default async (req: Request, res: Response) => {
    const securityService: MashroomSecurityService = req.pluginContext.services.secu

    // Create a permission
    await securityService.updateResourcePermission(req, {
```

```
        type: 'Page',
        key: pageId,
        permissions: [{
            permissions: ['View'],
            roles: ['Role1', 'Role2']
        }]
    });

    // Check a permission
    const mayAccess = await securityService.checkResourcePermission(req, 'Page', pag

    // ...
}
```

Services

MashroomSecurityService

The exposed service is accessible through *pluginContext.services.security.service*

Interface:

```
export interface MashroomSecurityService {
    /**
     * Get the current user or null if the user is not authenticated
     */
    getUser(request: Request): MashroomSecurityUser | null | undefined;

    /**
     * Checks if user != null
     */
    isAuthenticated(request: Request): boolean;

    /**
     * Check if the currently authenticated user has given role
     */
    isInRole(request: Request, roleName: string): boolean;

    /**
     * Check if the currently authenticated user is an admin (has the role Administr
     */
    isAdmin(request: Request): boolean;

    /**
     * Check the request against the ACL
     */
    checkACL(request: Request): Promise<boolean>;

    /**
```

```
     * Check if given abstract "resource" is permitted for currently authenticated u
     * The permission has to be defined with updateResourcePermission() first, other
     */
    checkResourcePermission(request: Request, resourceType: MashroomSecurityResource

    /**
     * Set a resource permission for given abstract resource.
     * A resource could be: {type: 'Page', key: 'home', permissions: [{ roles: ['Use
     *
     * If you pass a permission with an empty roles array it actually gets removed f
     */
    updateResourcePermission(request: Request, resource: MashroomSecurityProtectedRe

    /**
     * Get the permission definition for given resource, if any.
     */
    getResourcePermissions(request: Request, resourceType: MashroomSecurityResourceT

    /**
     * Add a role definition
     */
    addRoleDefinition(request: Request, roleDefinition: MashroomSecurityRoleDefiniti

    /**
     * Get all known roles. Returns all roles added with addRoleDefinition() or impl
     */
    getExistingRoles(request: Request): Promise<Array<MashroomSecurityRoleDefinitior

    /**
     * Check if an auto login would be possible.
     */
    canAuthenticateWithoutUserInteraction(request: Request): Promise<boolean>;

    /**
     * Start authentication process
     */
    authenticate(request: Request, response: Response): Promise<MashroomSecurityAuth

    /**
     * Check the existing authentication (if any)
     */
    checkAuthentication(request: Request): Promise<void>;

    /**
     * Get the authentication expiration time in unix time ms
     */
    getAuthenticationExpiration(request: Request): number | null | undefined;

    /**
     * Revoke the current authentication
```

```
     */
    revokeAuthentication(request: Request): Promise<void>;

    /**
     * Login user with given credentials (for form login).
     */
    login(request: Request, username: string, password: string): Promise<MashroomSec

    /**
     * Find a security provider by name.
     * Useful if you want to dispatch the authentication to a different provider.
     */
    getSecurityProvider(name: string): MashroomSecurityProvider | null | undefined;
}
```

Plugin Types

security-provider

This plugin type is responsible for the actual authentication and for creating a user object with a list of roles.

To register your custom security-provider plugin add this to *package.json*:

```
{
    "mashroom": {
        "plugins": [
            {
                "name": "My Custom Security Provider",
                "type": "security-provider",
                "bootstrap": "./dist/mashroom-bootstrap.js",
                "defaultConfig": {
                   "myProperty": "foo"
                }
            }
        ]
    }
}
```

The bootstrap returns the provider:

```
import type {MashroomSecurityProviderPluginBootstrapFunction} from '@mashroom/mashrc

const bootstrap: MashroomSecurityProviderPluginBootstrapFunction = async (pluginName

    return new MySecurityProvider(/* ... */);
};
```

```
export default bootstrap;
```

The provider has to implement the following interface:

```
export interface MashroomSecurityProvider {
    /**
     * Check if an auto login would be possible.
     * This is used for public pages when an authentication is optional but neverthe
     * It is safe to always return false.
     */
    canAuthenticateWithoutUserInteraction(request: Request): Promise<boolean>;
    /**
     * Start authentication process.
     * This typically means to redirect to the login page, then you should return st
     * This method could also automatically login the user, then you should return s
     */
    authenticate(request: Request, response: Response, authenticationHints?: any): P

    /**
     * Check the existing authentication (if any).
     * Use this to extend the authentication expiration or to periodically refresh t
     *
     * This method gets called for almost every request, so do nothing expensive her
     */
    checkAuthentication(request: Request): Promise<void>;

    /**
     * Get the authentication expiration time in unix time ms. Return null/undefinec
     */
    getAuthenticationExpiration(request: Request): number | null | undefined;

    /**
     * Revoke the current authentication.
     * That typically means to remove the user object from the session.
     */
    revokeAuthentication(request: Request): Promise<void>;

    /**
     * Programmatically login user with given credentials (optional, but necessary i
     */
    login(request: Request, username: string, password: string): Promise<MashroomSec

    /**
     * Get the current user or null if the user is not authenticated
     */
    getUser(request: Request): MashroomSecurityUser | null | undefined;
}
```

Mashroom Security Simple Provider

This plugin adds a simple, JSON file based security provider.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-security-provider-simple** as *dependency*.

To activate this provider configure the *Mashroom Security* plugin like this:

```
{
    "plugins": {
        "Mashroom Security Services": {
            "provider": "Mashroom Security Simple Provider"
        }
    }
}
```

And configure this plugin like this in the Mashroom config file:

```
{
  "plugins": {
        "Mashroom Security Simple Provider": {
            "users": "./users.json",
            "loginPage": "/login",
            "authenticationTimeoutSec": 1200
        }
    }
}
```

- *users*: The path to the JSON file with user and role definitions (Default: ./users.json)

- *loginPage*: The path to redirect to if a restricted resource is requested but the user not logged in yet (Default: /login)

- *authenticationTimeoutSec*: The inactivity time after that the authentication expires. Since this plugin uses the session to store make sure the session *cookie.maxAge* is greater than this value (Default: 1200)

  The content of the JSON file might look like this.

```
{
    "$schema": "https://www.mashroom-server.com/schemas/mashroom-security-simple-prc
    "users": [
        {
            "username": "admin",
            "displayName": "Administrator",
            "email": "xxxxx@xxxx.com",
            "pictureUrl": "xxxx",
            "passwordHash": "8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2
            "extraData": {
                "firstName": "John",
                "lastName": "Do"
            },
            "roles": [
                "Administrator"
            ],
            "secrets": {
                "token": "xxxxxxx"
            }
        },
        {
            "username": "john",
            "displayName": "John Do",
            "passwordHash": "96d9632f363564cc3032521409cf22a852f2032eec099ed5967c0d0
            "roles": [
                "User",
                "PowerUser"
            ]
        }
    ]
}
```

The *passwordHash* is the SHA256 hash of the password. *displayName*, *email* and *pictureUrl* are optional. *extraData* will be mapped to *user.extraData* and *secrets* will be mapped to *user.secrets*.

Mashroom LDAP Security Provider

This plugin adds an LDAP security provider.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-security-provider-ldap** as *dependency*.

To activate this provider, configure the *Mashroom Security* plugin like this:

```
{
    "plugins": {
```

```
        "Mashroom Security Services": {
            "provider": "Mashroom LDAP Security Provider"
        }
    }
}
```

And configure this plugin like this in the Mashroom config file:

```
{
    "plugins": {
        "Mashroom LDAP Security Provider": {
            "loginPage": "/login",
            "serverUrl": "ldap://my-ldap-server:636",
            "ldapConnectTimeout": 3000,
            "ldapTimeout": 5000,
            "bindDN": "uid=mashroom,dc=nonblocking,dc=at",
            "bindCredentials": "secret",
            "baseDN": "ou=users,dc=nonblocking,dc=at",
            "userSearchFilter": "(&(objectClass=person)(uid=@username@))",
            "groupSearchFilter": "(objectClass=group)",
            "extraDataMapping": {
                "mobile": "mobile",
                "address": "postalAddress"
            },
            "secretsMapping": {
                "internalUserId": "uid"
            },
            "groupToRoleMapping": "./groupToRoleMapping.json",
            "userToRoleMapping": "./userToRoleMapping.json",
            "authenticationTimeoutSec": 1200
        }
    }
}
```

- *loginPage*: The login URL to redirect to if the user is not authenticated (Default: /login)
- *serverUrl*: The LDAP server URL with protocol and port
- *ldapConnectTimeout*: Connect timeout in ms (Default: 3000)
- *ldapTimeout*: Timeout in ms (Default: 5000)
- *tlsOptions*: Optional TLS options if your LDAP server requires TLS. The options are passed to Node TLS but the file paths (e.g. for "cert") are

resolved relatively to the server config.

- *bindDN*: The bind user for searching
- *bindCredentials*: The password for the bind user
- *baseDN*: The base DN for searches (can be empty)
- *userSearchFilter*: The user search filter, *@username@* will be replaced by the actual username entered in the login form
- *groupSearchFilter*: The group search filter (can be empty if you don't want to fetch the user groups)
- *extraDataMapping*: Optionally map extra LDAP attributes to *user.extraData*. The key in the map is the extraData property, the value the LDAP attribute (Default: null)
- *secretsMapping*: Optionally map extra LDAP attributes to *user.secrets* (Default: null)
- *groupToRoleMapping*: An optional JSON file that contains a user group to roles mapping (Default: /groupToRoleMapping.json)
- *userToRoleMapping*: An optional JSON file that contains a user name to roles mapping (Default: /userToRoleMapping.json)
- *authenticationTimeoutSec*: The inactivity time after that the authentication expires. Since this plugin uses the session to store make sure the session *cookie.maxAge* is greater than this value (Default: 1200)

For a server that requires TLS you have to provide a *tlsOptions* object:

```
{
    "plugins": {
        "Mashroom LDAP Security Provider": {
            "serverUrl": "ldaps://my-ldap-server:636",
            "tlsOptions": {
              "cert": "./server-cert.pem",

              // Necessary only if the server requires client certificate authentica
              //"key": "./client-key.pem",

              // Necessary only if the server uses a self-signed certificate.
              // "rejectUnauthorized": false,
```

```
            // "ca": [ "./server-cert.pem" ],
        }
    }
}
}
```

The *groupToRoleMapping* file has to following simple structure:

```
{
    "$schema": "https://www.mashroom-server.com/schemas/mashroom-security-ldap-provi
    "LDAP_GROUP1": [
        "ROLE1",
        "ROLE2"
    ]
}
```

And the *userToRoleMapping* file:

```
{
    "$schema": "https://www.mashroom-server.com/schemas/mashroom-security-ldap-provi
    "username": [
        "ROLE1",
        "ROLE2"
    ]
}
```

Mashroom OpenID Connect Security Provider

This plugin adds an OpenID Connect/OAuth2 security provider that can be used to integrate *Mashroom Server* with almost all Identity Providers or Identity Platforms.

Tested with:

- Github OAuth 2.0
- Google Identity Platform
- Keylcoak Identity Provider
- Open AM

Should work with (among others):

- Auth0

- Facebook Login
- Gluu
- Microsoft Identity Platform
- Okta
- OneLogin
- PingIdentity

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-security-provider-ldap** as *dependency*.

To activate this provider, configure the *Mashroom Security* plugin like this:

```
{
    "plugins": {
        "Mashroom Security Services": {
            "provider": "Mashroom OpenID Connect Security Provider"
        }
    }
}
```

And configure this plugin like this in the Mashroom config file:

```
{
    "plugins": {
        "Mashroom OpenID Connect Security Provider": {
            "mode": "OIDC",
            "issuerDiscoveryUrl": "http://localhost:8080/.well-known/openid-configur
            "issuerMetadata": null,
            "scope": "openid email profile",
            "clientId": "mashroom",
            "clientSecret": "your-client-secret",
            "redirectUrl": "http://localhost:5050/openid-connect-cb",
            "responseType": "code",
            "usePKCE": false,
            "extraAuthParams": {},
            "extraDataMapping": {
                "phone": "phone",
                "birthdate": "birthdate",
                "updatedAt": "updated_at"
```

```
            },
            "rolesClaimName": "roles",
            "adminRoles": [
                "mashroom-admin"
            ],
            "httpRequestTimeoutMs": 3500
        },
        "Mashroom OpenID Connect Security Provider Callback": {
           "path": "/openid-connect-cb"
        }
    }
}
```

- *mode*: Can be *OIDC* (default) or *OAuth2*. Pure OAuth2 usually does not support permission roles (for authorization).
- *issuerDiscoveryUrl*: The OpenID Connect Discovery URL , this is usually https://<your-idp-host>/.well-known/openid-configuration. See Example Configurations below.
- *issuerMetadata*: The issuer metadata if no *issuerDiscoveryUrl* is available. Will be passed to the Issuer constructor . See examples below.
- *scope*: The scope (permissions) to ask for (Default: openid email profile)
- *clientId*: The client to use (Default: mashroom)
- *clientSecret*: The client secret
- *redirectUrl*: The full URL of the callback (as seen from the user). This is usually https://<mashroom-server-host>/openid-connect-cb. The path corresponds with the *path* property in the *Mashroom OpenID Connect Security Provider Callback* config.
- *responseType*: The OpenID Connect response type (flow) to use (Default: code)
- *usePKCE*: Use the Proof Key for Code Exchange extension for the *code* flow (Default: false)
- *extraAuthParams*: Extra authentication parameters that should be used
- *extraDataMapping*: Optionally map extra claims to *user.extraData*. The key in the map is the extraData property, the value the claim name (Default: null)

- *rolesClaimName*: Defines the name of the claim (the property of the claims or userinfo object) that contains the user roles array (Default: roles)
- *adminRoles*: A list of user roles that should get the Mashroom *Administrator* role (Default: ["mashroom-admin"])
- *httpRequestTimeoutMs*: Request timeout when contacting the Authorization Server (Default: 3500)

Roles

Since the authorization mechanism relies on user roles it is necessary to configure your identity provider to map the user roles to a scope (which means we can get it as claim). See Example Configurations below.

Secrets

The plugin maps the ID/JWT Token to *user.secrets.idToken* so it can for example be used in a Http Proxy Interceptor to set the Bearer for backend calls.

Authentication Expiration

The implementation automatically extends the authentication via refresh token every view seconds (as long as the user is active). So, if the authentication session gets revoked in the identity provider the user is signed out almost immediately.

The expiration time of the access token defines after which time the user is automatically signed out due to inactivity. And the expiration time of the refresh token defines how long the user can work without signing in again.

Example Configurations

Keycloak

Setup:

- Create a new client in your realm (e.g. *mashroom*)
- In the *Settings* tab set Access Type *confidential*
- Make sure the *Valid Redirect URIs* contain your redirect URL (e.g. http://localhost:5050/*)
- In the *Credentials* tab you'll find the client secret

- To map the roles to a scope/claim goto *Mappers*, click *Add Builtin* and add a *realm roles* mapper. In the field *Token Claim Name* enter *roles*. Also check *Add to ID token*.

- You should create a role (e.g. *mashroom-admin*) for users with Administrator rights

You'll find more details about the configuration here:
https://www.keycloak.org/documentation.html

If your Keycloak runs on localhost, the Realm name is *test* amd the client name *mashroom*, then the config would look like this:

```
{
    "plugins": {
        "Mashroom OpenID Connect Security Provider": {
            "issuerDiscoveryUrl": "http://localhost:8080/auth/realms/test/.well-know
            "clientId": "mashroom",
            "clientSecret": "xxxxxxxxxxxxxxx",
            "redirectUrl": "http://localhost:5050/openid-connect-cb",
            "rolesClaim": "roles",
            "adminRoles": [
                "mashroom-admin"
            ]
        }
    }
}
```

OpenAM

Setup:

- Create a new Realm (e.g. *Test*)

- Create a new OIDC configuration: OAuth provider -> Configure OpenID Connect -> Create from the Dashboard)

- Create a new Agent (e.g. *mashroom*): Applications -> OAuth 2.0 -> Agent from the Dashboard)

- Make sure the agent has at least the scopes *openid email profile* and the *ID Token Signing Algorithm* is set to *RS256*

- Follow this KB article to add the OpenAM groups as roles claim

You'll find more details about the configuration here:
https://backstage.forgerock.com/docs/openam/13.5/admin-guide

If your OpenAM server runs on localhost, the Realm name is *Test* amd the client name *mashroom*, then the config would look like this:

```
{
    "plugins": {
        "Mashroom OpenID Connect Security Provider": {
            "issuerDiscoveryUrl": "http://localhost:8080/openam/oauth2/Test/.well-kn
            "scope": "openid email profile",
            "clientId": "mashroom",
            "clientSecret": "mashroom",
            "redirectUrl": "http://localhost:5050/openid-connect-cb",
            "rolesClaim": "roles",
            "adminRoles": [
                "mashroom-admin"
            ]
        }
    }
}
```

Google Identity Platform

Setup:

- Go to: https://console.developers.google.com
- Select *Credentials* from the menu and then the auto created client under *OAuth 2.0 Client IDs*
- Make sure the *Authorized* redirect URIs contains your redirect URL (e.g. http://localhost:5050/openid-connect-cb)
- Create a *OAuth consent screen*

Possible config:

```
{
    "plugins": {
        "Mashroom OpenID Connect Security Provider": {
            "issuerDiscoveryUrl": "https://accounts.google.com/.well-known/openid-co
            "scope": "openid email profile",
            "clientId": "xxxxxxxxxxxxxxx.apps.googleusercontent.com",
            "clientSecret": "xxxxxxxxxxxxxxx",
            "redirectUrl": "http://localhost:5050/openid-connect-cb",
```

```
            "extraAuthParams": {
                "access_type": "offline"
            },
            "usePKCE": true
        }
    }
}
```

The *access_type=offline* parameter is necessary to get a refresh token.

Since Google users don't have authorization roles there is no way to make some users *Administrator*.

GitHub OAuth2

Setup:

- Go to: https://github.com/settings/developers
- Click on "New OAuth App"
- Enter the application name and correct callback URL (e.g. http://localhost:5050/openid-connect-cb)

Possible config:

```
{
    "plugins": {
        "Mashroom OpenID Connect Security Provider": {
            "mode": "OAuth2",
            "issuerMetadata": {
                "issuer": "GitHub",
                "authorization_endpoint": "https://github.com/login/oauth/authorize"
                "token_endpoint": "https://github.com/login/oauth/access_token",
                "userinfo_endpoint": "https://api.github.com/user",
                "end_session_endpoint": null
            },
            "scope": "openid email profile",
            "clientId": "xxxxxxxxxxxxxxxx",
            "clientSecret": "xxxxxxxxxxxxxxxx",
            "redirectUrl": "http://localhost:5050/openid-connect-cb"
        }
    }
}
```

Since GitHub uses pure OAuth2 the users don't have permission roles and there is no way to make some users *Administrator*. It also supports no *userinfo* endpoint, so it actually makes not

much sense to use it with *Mashroom*.

Mashroom Basic Authentication Wrapper Security Provider

This plugin adds support for Basic authentication to any other security provider that implements *login()* properly. This can be useful when you need to access some APIs on the server from an external system or for test purposes.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-security-provider-basic-wrapper** as *dependency*.

To activate this provider configure the *Mashroom Security* plugin like this:

```
{
    "plugins": {
        "Mashroom Security Services": {
            "provider": "Mashroom Basic Wrapper Security Provider"
        }
    }
}
```

And configure this plugin like this in the Mashroom config file:

```
{
  "plugins": {
        "Mashroom Basic Wrapper Security Provider": {
             "targetSecurityProvider": "Mashroom Security Simple Provider",
             "onlyPreemptive": true,
             "realm": "mashroom"
        }
    }
}
```

- *targetSecurityProvider*: The actual security provider that is used to login (Default: Mashroom Security Simple Provider)
- *onlyPreemptive*: Only use BASIC if it is sent preemptively if true. Otherwise, the plugin will send HTTP 401 and *WWW-Authenticate* which will trigger the Browser's login popup (Default: true)
- *realm*: The realm name that should be used if *onlyPreemptive* is false (Default: mashroom)

Mashroom Security Default Login Webapp

This plugin adds a default login webapp which can be used for security providers that require a login page.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-security-default-login-webapp** as *dependency*.

You can override the default config in your Mashroom config file like this:

```
{
  "plugins": {
        "Mashroom Security Default Login Webapp": {
            "path": "/login",
            "pageTitle": "My fancy website",
            "loginFormTitle": "Login",
            "styleFile": "./login_style.css"
        }
    }
}
```

- *path*: The path of the login page (Default: /login)
- *pageTitle*: A custom page title, can be the actual title or a message key (i18n) (Default is the server name)
- *loginFormTitle*: A custom title for the login form, can be the actual title or a message key (i18n) (Default: *login*)
- *styleFile*: Custom CSS that will be loaded instead of the built-in style (relative to Mashroom config file, default: null)

Mashroom CSRF Protection

If you add this plugin all updating HTTP methods (such as POST, PUT and DELETE) must contain a CSRF token automatically generated for the session. Otherwise, the request will be rejected.

There are two ways to pass the token:

- As HTTP header *X-CSRF-Token*
- As query parameter *csrfToken*

You can use the *MashroomCSRFService* to get the current token.

Mashroom Portal automatically uses this plugin to secure all requests if available.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-csrf-protection** as *dependency*.

After that you can use the service like this:

```
import type {MashroomCacheControlService} from '@mashroom/mashroom-csrf-protection/t

export default (req: Request, res: Response) => {

    const csrfService: MashroomCacheControlService = req.pluginContext.services.csrf
    const token = csrfService.getCSRFToken(req);

    // ...
}
```

You can override the default config in your Mashroom config file like this:

```
{
  "plugins": {
        "Mashroom CSRF Middleware": {
            "safeMethods": ["GET", "HEAD", "OPTIONS"]
        },
        "Mashroom CSRF Services": {
            "saltLength": 8,
            "secretLength": 18
        }
    }
}
```

- *safeMethods*: List of HTTP methods that require no CSRF token check (Default: ["GET", "HEAD", "OPTIONS"]).
- *saltLength* and *secretLength* are passed to the csrf package.

Services

MashroomCSRFService

The exposed service is accessible through *pluginContext.services.csrf.service*

Interface:

```
export interface MashroomCSRFService {

    /**
```

```
     * Get the current CSRF token for this session
     */
    getCSRFToken(request: Request): string;

    /**
     * Check if the given token is valid
     */
    isValidCSRFToken(request: Request, token: string): boolean;
}
```

Mashroom Helmet

This plugin adds the Helmet middleware which sets a bunch of protective HTTP headers on each response.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-helmet** as *dependency*.

You can override the default config in your Mashroom config file like this:

```
{
    "plugins": {
        "Mashroom Helmet Middleware": {
            "helmet": {
                "contentSecurityPolicy": false,
                "crossOriginEmbedderPolicy": false,
                "crossOriginOpenerPolicy": {
                    "policy": "same-origin"
                },
                "crossOriginResourcePolicy": {
                    "policy": "same-site"
                },
                "expectCt": false,
                "referrerPolicy": false,
                "hsts": {
                    "maxAge": 31536000
                },
                "noSniff": true,
                "originAgentCluster": false,
                "dnsPrefetchControl": {
                    "allow": false
                },
                "frameguard": {
                    "action": "sameorigin"
                },
                "permittedCrossDomainPolicies": {
                    "permittedPolicies": "none"
                },
```

```
                "hidePoweredBy": false,
                "xssFilter": true
            }
        }
    }
}
```

- *helmet*: The configuration will directly be passed to *Helmet* middelware. Checkout the Helmet Documentation for available options.

NOTE: You shouldn't enable the *noCache* module because this would significantly decrease the performance of the *Mashroom Portal*.

Mashroom Error Pages

This plugin allows it to show proper HTML pages for arbitrary HTTP status codes. It delivers error pages only if the request accept header contains text/html. So, typically not for AJAX requests.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-error-pages** as *dependency*.

You can override the default config in your Mashroom config file like this:

```
{
  "plugins": {
        "Mashroom Error Pages Middleware": {
            "mapping": {
                "404": "./pages/404.html",
                "403": "./pages/403.html",
                "400": "http://my.server-com/bad_request.html",
                "500": "/some/server/path/500.html",
                "default": "./pages/default.html"
            }
        }
    }
}
```

- *mapping*: Maps status codes to error pages. The target files can be file paths or HTTP/S urls. If the file path is not absolute the plugin will expect it to be relative to the plugin folder or the Mashroom server config file. If a status code is not defined in the mapping or no default exists, no error page will be shown.

- The HTML files should not reference *local* resources (Images, CSS, JavaScript) because they cannot be loaded

- They may contain the following placeholders:
  - *$REQUEST*URL_: The original request URL
  - *$STATUS*CODE_: The status code
  - *$MASHROOM*VERSION_: The _Mashroom Server_ version
  - *$MESSAGE[messageKey(,Default text if i18n not yet available)]*: A translated message from the *mashroom-i18n* package

Mashroom Storage

This plugin adds a storage service abstraction that delegates to a provider plugin.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-storage** as *dependency*.

Then use the storage service like this:

```
import type {MashroomStorageService} from '@mashroom/mashroom-storage/type-definitic

export default async (req: Request, res: ExpressResponse) => {
    const storageService: MashroomStorageService = req.pluginContext.services.storag

    const customerCollection = await storageService.getCollection('my-collection');

    const customer = await customerCollection.findOne({customerNr: 1234567});
    const customers = await customerCollection.find({ $and: [{ name: { $regex: 'jo.*

    // ...
}
```

You can override the default config in your Mashroom config file like this:

```
{
  "plugins": {
       "Mashroom Storage Services": {
           "provider": "Mashroom Storage Filestore Provider",
           "memoryCache": {
               "enabled": false,
               "ttlSec": 120,
               "invalidateOnUpdate": true,
               "collections": {
```

```
                    "mashroom-portal-pages": {
                        "enabled": true,
                        "ttlSec": 300
                    }
                }
            }
        }
    }
}
```

- *provider*: The storage-provider plugin that implements the actual storage (Default: Mashroom Storage Filestore Provider)

- *memoryCache*: Use the memory cache to improve the performance. Requires *@mashroom/mashroom-memory-cache* to be installed.
  - *enabled*: Enable cache (of all) collections. The preferred way is to set this to false and enable caching per collection (Default: false)
  - *ttlSec*: The default TTL in seconds. Can be overwritten per collection (Default: 120)
  - *invalidateOnUpdate*: Clear the cache for the whole collection if an entry gets updated (Default: true). This might be an expensive operation on some memory cache implementations (e.g. based on Redis). So use this only if updates don't happen frequently.
  - *collections*: A map of collections specific settings. You can overwrite here *enabled*, *ttlSec* and *invalidateOnUpdate*.

Services

MashroomStorageService

The exposed service is accessible through *pluginContext.services.storage.service*

Interface:

```
export interface MashroomStorageService {
    /**
     * Get (or create) the MashroomStorageCollection with given name.
     */
    getCollection<T extends StorageRecord>(name: string): Promise<MashroomStorageCol
}

export interface MashroomStorageCollection<T extends MashroomStorageRecord> {
    /**
     * Find all items that match given filter. The filter supports a subset of Mongo
     */
    find(filter?: MashroomStorageObjectFilter<T>, limit?: number, skip?: number, sor
```

```
    /**
     * Return the first item that matches the given filter or null otherwise.
     */
    findOne(filter: MashroomStorageObjectFilter<T>): Promise<MashroomStorageObject<T

    /**
     * Insert one item
     */
    insertOne(item: T): Promise<MashroomStorageObject<T>>;

    /**
     * Update the first item that matches the given filter.
     */
    updateOne(filter: MashroomStorageObjectFilter<T>, propertiesToUpdate: Partial<Ma

    /**
     * Update multiple entries
     */
    updateMany(filter: MashroomStorageObjectFilter<T>, propertiesToUpdate: Partial<M

    /**
     * Replace the first item that matches the given filter.
     */
    replaceOne(filter: MashroomStorageObjectFilter<T>, newItem: T): Promise<Mashroom

    /**
     * Delete the first item that matches the given filter.
     */
    deleteOne(filter: MashroomStorageObjectFilter<T>): Promise<MashroomStorageDelete

    /**
     * Delete all items that match the given filter.
     */
    deleteMany(filter: MashroomStorageObjectFilter<T>): Promise<MashroomStorageDelet
}
```

Plugin type

storage-provider

This plugin type adds a a new storage implementation that can be used by this plugin.

To register a new storage-provider plugin add this to *package.json*:

```
{
    "mashroom": {
        "plugins": [
            {
                "name": "My Storage Provider",
```

```
                "type": "storage-provider",
                "bootstrap": "./dist/mashroom-bootstrap.js",
                "defaultConfig": {
                    "myProperty": "test"
                }
            }
        ]
    }
}
```

The bootstrap returns the provider:

```
import MyStorage from './MyStorage';

import type {MashroomStoragePluginBootstrapFunction} from '@mashroom/mashroom-storag

const bootstrap: MashroomStoragePluginBootstrapFunction = async (pluginName, pluginC

    return new MyStorage(/* .... */);
};

export default bootstrap;
```

The plugin has to implement the following interfaces:

```
export interface MashroomStorage {
    /**
     * Get (or create) the MashroomStorageCollection with given name.
     */
    getCollection<T extends StorageRecord>(
        name: string,
    ): Promise<MashroomStorageCollection<T>>;
}
```

Mashroom Storage Filestore Provider

This plugin adds a simple but cluster-safe, JSON based storage provider.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-storage-provider-filestore** as *dependency*.

To activate this provider configure the *Mashroom Security* plugin like this:

```
{
    "plugins": {
         "Mashroom Storage Services": {
```

```
            "provider": "Mashroom Storage Filestore Provider"
        }
    }
}
```

And configure this plugin like this in the Mashroom config file:

```
{
  "plugins": {
        "Mashroom Storage Filestore Provider": {
            "dataFolder": "/var/mashroom/data/storage",
            "checkExternalChangePeriodMs": 100,
            "prettyPrintJson": true
        }
    }
}
```

- *dataFolder*: The **shared folder** to store the data files. The base for relative paths is the Mashroom config file (Default: ./data/storage)
- *checkExternalChangePeriodMs*: Poll interval for external file changes (by other servers in the cluster). You can increase the default if you run a single server, the config is readonly or performance is more important than consistency (Default: 100)
- *prettyPrintJson*: Pretty print the JSON files to make it human-readable (Default: true)

Mashroom Storage MongoDB Provider

This plugin adds a MongoDB based storage provider.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-storage-provider-mongodb** as *dependency*.

To activate this provider, configure the *Mashroom Security* plugin like this:

```
{
    "plugins": {
         "Mashroom Storage Services": {
             "provider": "Mashroom Storage MongoDB Provider"
         }
    }
}
```

And configure this plugin like this in the Mashroom config file:

```
{
  "plugins": {
        "Mashroom Storage MongoDB Provider": {
            "uri": "mongodb://user:xxxxx@localhost:27017/mashroom_storage_db",
            "connectionOptions": {
                "poolSize": 5,
                "useUnifiedTopology": true,
                "useNewUrlParser": true
            }
        }
    }
}
```

- *uri*: A MongoDB connection string (see MongoDB documentation). **Must** contain the database to use.
- *connectionOptions*: The MongoDB connection options (see MongoDB documentation).

Mashroom Session

This plugin adds Express session as middleware.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-session** as *dependency*.

You can override the default config in your Mashroom config file like this:

```
{
    "plugins": {
        "Mashroom Session Middleware": {
            "order": -100,
            "provider": "Mashroom Session Filestore Provider",
            "session": {
                "secret": "EWhQ5hvETGkqvPDA",
                "resave": false,
                "saveUninitialized": false,
                "cookie": {
                    "httpOnly": true,
                    "secure": false,
                    "sameSite": false
                }
            }
        }
```

```
    }
}
```

- *order*: The middleware order (Default: -100)
- *provider*: The plugin from type *session-store-provider* that implements the store (Default: memory)
- *session*: The properties are just passed to express-session. See Express session for possible options.
  - *cookie.maxAge*: Max cookie age in ms, which should be the max expected session duration (Default 2h)

Security hints:

- Change the default *secret*
- You should consider setting *cookie.sameSite* to either "lax" or "strict" (CSRF protection).

Plugin Types

session-store-provider

This plugin type adds a session store that can be used by this plugin.

To register a custom session-store-provider plugin add this to *package.json*:

```
{
    "mashroom": {
        "plugins": [
            {
                "name": "My Session Provider",
                "type": "session-store-provider",
                "bootstrap": "./dist/mashroom-bootstrap.js",
                "defaultConfig": {
                    "myProperty": "test"
                }
            }
        ]
    }
}
```

The bootstrap returns the express session store (here for example the file store):

```
import sessionFileStore from 'session-file-store';

import type {MashroomSessionStoreProviderPluginBootstrapFunction} from '@mashroom/ma

const bootstrap: MashroomSessionStoreProviderPluginBootstrapFunction = async (plugir
    const options = {...pluginConfig};
    const FileStore = sessionFileStore(expressSession);
    return new FileStore(options);
};

export default bootstrap;
```

Mashroom Session Filestore Provider

This plugin adds a file based session store that can be used by *Mashroom Session*. Actually this is just a wrapper for the session-file-store package.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-session-provider-filestore** as *dependency*.

Activate this session provider in your Mashroom config file like this:

```
{
  "plugins": {
        "Mashroom Session Middleware": {
            "provider": "Mashroom Session Filestore Provider"
        }
    }
}
```

And to change the default config of this plugin add:

```
{
  "plugins": {
        "Mashroom Session Filestore Provider": {
            "path": "../../data/sessions"
        }
    }
}
```

NOTE: The base for relative paths is the Mashroom config file.

All config options are passed to the *session-file-store*. See session-file-store for available options.

Mashroom Session Redis Provider

This plugin adds a Redis session store that can be used by *Mashroom Session*. Actually this is just a wrapper for the connect-redis package.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-session-provider-redis** as *dependency*.

Activate this session provider in your Mashroom config file like this:

```
{
  "plugins": {
        "Mashroom Session Redis Provider": {
            "provider": "Mashroom Session MongoDB Provider"
        }
    }
}
```

And to change the default config of this plugin add:

```
{
  "plugins": {
      "Mashroom Session Redis Provider": {
          "client": {
              "redisOptions": {
                  "host": "localhost",
                  "port": "6379",
                  "maxRetriesPerRequest": 3,
                  "enableOfflineQueue": false
              },
              "cluster": false,
              "clusterNodes": null,
              "clusterOptions": null
          },
          "prefix": "mashroom:sess:"
          "ttl": 86400
      }
  }
}
```

- *client*: Options for the Redis client. *redisOptions* are just to the *Redis* constructor of [ioredis](https://github.com/luin/ioredis Checkout out the ioredis documentation for all available options.

- *prefix*: The key prefix. Appends to whatever prefix you may have set on the client itself. (Default: mashroom:sess:)

- *ttl*: TTL in seconds (Default: 86400 - one day)

> **NOTE**: Don't set *client.redisOptions.keyPrefix* because then the session metrics will not work properly.

Usage with Sentinel

For a high availability cluster with Sentinel the configuration would look like this:

```
{
  "plugins": {
      "Mashroom Session Redis Provider": {
         "client": {
             "redisOptions": {
                 "sentinels": [
                     { "host": "localhost", "port": 26379 },
                     { "host": "localhost", "port": 26380 }
                 ],
                 "name": "myMaster",
                 "keyPrefix": "mashroom:sess:"
             }
         }
      }
  }
}
```

- *sentinels*: list of sentinel nodes to connect to

- *name*: identifies a group of Redis instances composed of a master and one or more slaves

Checkout out the *Sentinel* section of the ioredis documentation for all available options.

Usage with a cluster

For a sharding cluster configure the plugin like this:

```
{
  "plugins": {
      "Mashroom Session Redis Provider": {
```

```
        "client": {
            "cluster": true,
            "clusterNodes": [
                {
                    "host": "redis-node1",
                    "port": "6379"
                },
                {
                    "host": "redis-node2",
                    "port": "6379"
                }
            ],
            "clusterOptions": {
                "maxRedirections": 3
            },
            "redisOptions": {
                "keyPrefix": "mashroom:sess:"
            }
        }
    }
  }
}
```

- *cluster*: Enables cluster support, must be true
- *clusterNodes*: Cluster start nodes
- *clusterOptions*: Passed as second argument of the *Redis.Cluster* constructor of *ioredis*
- *redisOptions*: Passed as *redisOptions* in the *clusterOptions*

Checkout out the *Cluster* section of the ioredis documentation for all available options.

Mashroom Session MongoDB Provider

This plugin adds a mongoDB session store that can be used by *Mashroom Session*. Actually this is just a wrapper for the connect-mongo package.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-session-provider-mongodb** as *dependency*.

Activate this session provider in your Mashroom config file like this:

```
{
  "plugins": {
```

```
        "Mashroom Session Middleware": {
            "provider": "Mashroom Session MongoDB Provider"
        }
    }
}
```

And to change the default config of this plugin add:

```
{
  "plugins": {
      "Mashroom Session MongoDB Provider": {
          "client": {
              "uri": "mongodb://localhost:27017/mashroom_session_db?connectTimeoutMS
              "connectionOptions": {
                  "minPoolSize": 5,
                  "serverSelectionTimeoutMS": 3000
              }
          },
          "collectionName": "mashroom-sessions",
          "ttl": 86400,
          "autoRemove": "native",
          "autoRemoveInterval": 10,
          "touchAfter": 0,
          "crypto": {
              "secret": false
          }
      }
    }
}
```

- *client*: Options to construct the client. *connectionOptions* are passed to the mongodb driver .
- *collectionName*: Mongo collection to store sessions (Default: mashroom-sessions)
- *ttl*: TTL in seconds (Default: 86400 - one day)
- *autoRemove*: Session remove strategy (Default: native)
- *autoRemoveInterval*: Remove interval in seconds if *autoRemove* is interval (Default: 10)
- *touchAfter*: Interval in seconds between session updates (Default: 0)

- *crypto*: Options regarding session encryption. For details see connect-mongo .

Mashroom HTTP proxy

This plugin adds a service for forwarding requests to a target URI. It supports HTTP, HTTPS and WebSockets (only the default/nodeHttpProxy implementation, see below).

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-http-proxy** as *dependency*.

After that you can use the service like this:

```
import type {MashroomHttpProxyService} from '@mashroom/mashroom-http-proxy/type-defi

export default async (req: Request, res: Response) => {
    const httpProxyService: MashroomHttpProxyService = req.pluginContext.services.pr

    const targetURI = 'http://foo.bar/api/test';
    const additionalHeaders = {};
    await httpProxyService.forward(req, res, targetURI, additionalHeaders);
}
```

You can override the default config in your Mashroom config file like this:

```
{
    "plugins": {
        "Mashroom Http Proxy Services": {
            "forwardMethods": [
                "GET",
                "POST",
                "PUT",
                "DELETE"
            ],
            "forwardHeaders": [
              "accept",
              "accept-*",
              "range",
              "expires",
              "cache-control",
              "last-modified",
              "content-*",
              "x-forwarded-*",
              "uber-trace-id",
              "uberctx-",
              "b3",
```

```
            "x-b3-*",
            "trace*",
            "sec-websocket-*"
          ],
          "rejectUnauthorized": true,
          "poolMaxTotalSockets": null,
          "poolMaxSocketsPerHost": 10,
          "poolMaxWaitingRequestsPerHost": null,
          "socketTimeoutMs": 30000,
          "keepAlive": true,
          "retryOnReset": true,
          "wsMaxConnectionsTotal": 2000,
          "wsMaxConnectionsPerHost": null,
          "proxyImpl": "default"
        }
    }
}
```

- *forwardMethods*: The HTTP methods that should be forwarded
- *forwardHeaders*: The HTTP headers that should be forwarded. May contain a * as wildcard.
- *rejectUnauthorized*: Reject self-signed certificates (Default: true)
- *poolMaxTotalSockets*: Max HTTP pool sockets total (Default: null - no limit)
- *poolMaxSocketsPerHost*: Max HTTP pool sockets per target host (Default: 10)
- *poolMaxWaitingRequestsPerHost*: Max waiting HTTP requests per target host, needs to be > 0 if set (Default: null - no limit)
- *socketTimeoutMs*: HTTP socket timeout, which is the time the target has to accept the connection and start sending the response (Default: 30000)
- *keepAlive*: HTTP connection keep-alive. Set this to *false* if you experience random ECONNRESET with the *nodeHttpProxy* implementation, see: https://github.com/nonblocking/mashroom/issues/77 (Default: true)
- *retryOnReset*: If the target resets the HTTP connection (because a keep-alive connection is broken) retry once (Default: true)
- *wsMaxConnectionsTotal*: Max WebSocket connections total. Set this to 0 if

you want to disable the WS proxy (Default: 2000)

- *wsMaxConnectionsPerHost*: Max WebSocket connections per target host (Default: 0 - no limit)
- *createForwardedForHeaders*: Add *x-forwarded-* headers to the outgoing request (Default: false)
- *proxyImpl*: Switch the proxy implementation. Currently available are:
- *streamAPI* (based on the Node.js stream API)
- *nodeHttpProxy* (based on node-http-proxy)
- *default* (which is *streamAPI*)

Services

MashroomHttpProxyService

The exposed service is accessible through *pluginContext.services.proxy.service*

Interface:

```
export interface MashroomHttpProxyService {

    /**
     * Forwards the given request to the targetUri and passes the response from the
     * The Promise will always resolve, you have to check response.statusCode to see
     * The Promise will resolve as soon as the whole response was sent to the client
     */
    forward(req: Request, res: Response, targetUri: string, additionalHeaders?: Http

    /**
     * Forwards a WebSocket request (ws or wss).
     * The passed additional headers are only available at the upgrade/handshake req
     */
    forwardWs(req: IncomingMessageWithContext, socket: Socket, head: Buffer, targetU

}
```

Plugin Types

http-proxy-interceptor

This plugin type can be used to intercept http proxy calls and to add for example authentication headers to backend calls.

To register your custom http-proxy-interceptor plugin add this to *package.json*:

```
{
    "mashroom": {
        "plugins": [
            {
                "name": "My Custom Http Proxy Interceptor",
                "type": "http-proxy-interceptor",
                "bootstrap": "./dist/mashroom-bootstrap.js",
                "defaultConfig": {
                    "order": 500,
                    "myProperty": "foo"
                }
            }
        ]
    }
}
```

- *defaultConfig.order*: The weight of the middleware in the stack - the higher it is the **later** it will be executed (Default: 1000)

The bootstrap returns the interceptor:

```
import type {MashroomHttpProxyInterceptorPluginBootstrapFunction} from '@mashroom/ma

const bootstrap: MashroomHttpProxyInterceptorPluginBootstrapFunction = async (plugir

    return new MyInterceptor(/* ... */);
};

export default bootstrap;
```

The provider has to implement the following interface:

```
interface MashroomHttpProxyInterceptor {



    /**
     * Intercept request to given targetUri.
     *
     * The existingHeaders contain the original request headers, headers added by th
     * The existingQueryParams contain query parameters from the request and the one
     *
     * clientRequest is the request that shall be forwarded. DO NOT MANIPULATE IT. J
```

```
     *
     * Return null or undefined if you don't want to interfere with a call.
     */
    interceptRequest?(targetUri: string, existingHeaders: Readonly<HttpHeaders>, exi
                      clientRequest: Request, clientResponse: Response):
        Promise<MashroomHttpProxyRequestInterceptorResult | undefined | null>;

    /**
     * Intercept WebSocket request to given targetUri.
     *
     * The existingHeaders contain the original request headers, headers added by th
     *
     * The changes are ONLY applied to the upgrade request, not to WebSocket message
     *
     * Return null or undefined if you don't want to interfere with a call.
     */
    interceptWsRequest?(targetUri: string, existingHeaders: Readonly<HttpHeaders>, c
        Promise<MashroomWsProxyRequestInterceptorResult | undefined | null>;

    /**
     * Intercept response from given targetUri.
     *
     * The existingHeaders contain the original request header and the ones already
     * targetResponse is the response that shall be forwarded to the client. DO NOT
     *
     * Return null or undefined if you don't want to interfere with a call.
     */
    interceptResponse?(targetUri: string, existingHeaders: Readonly<HttpHeaders>, ta
                       clientRequest: Request, clientResponse: Response):
        Promise<MashroomHttpProxyResponseInterceptorResult | undefined | null>;

}
```

Examples

Add a Bearer token to each request like this (implemented like this in the *mashroom-http-proxy-add-id-token* module):

```
export default class MyInterceptor implements MashroomHttpProxyInterceptor {

    async interceptRequest(targetUri: string, existingHeaders: Readonly<HttpHeaders>
                           clientRequest: Request, clientResponse: Response) {
        const logger = clientRequest.pluginContext.loggerFactory('test.http.interce
        const securityService = clientRequest.pluginContext.services.security && cli

        const user = securityService.getUser(clientRequest);
        if (!user) {
           return;
```

```
        }

        return {
           addHeaders: {
              Authorization: `Bearer ${user.secrets.idToken}`
           }
        };
    }
}
```

Return forbidden for some reason:

```
export default class MyInterceptor implements MashroomHttpProxyInterceptor {

    async interceptRequest(targetUri: string, existingHeaders: Readonly<HttpHeaders>
                           clientRequest: Request, clientResponse: Response) {

        clientResponse.sendStatus(403);

        return {
          responseHandled: true
        };
    }
}
```

Handle the response instead of the proxy:

```
export default class MyInterceptor implements MashroomHttpProxyInterceptor {

    async interceptResponse(targetUri: string, existingHeaders: Readonly<HttpHeaders
        let body = [];
        targetResponse.on('data', function (chunk) {
            body.push(chunk);
        });
        targetResponse.on('end', function () {
            body = Buffer.concat(body).toString();
            console.log("Response from proxied server:", body);
            clientResponse.json({ success: true });
        });

        // NOTE: if you "await" the end event you have to call targetResponse.resume
        //  because the interceptor pauses the stream from the target until all inte

        return {
            responseHandled: true
        };
```

```
    }
}
```

Compress request/response body (only supported by the default/*streamAPI* proxy implementation):

```
import zlib from 'zlib';

export default class MyInterceptor implements MashroomHttpProxyInterceptor {

    async interceptRequest(targetUri) {
        if (targetUri.startsWith('https://my-backend-server.com')) {
            return {
                addHeaders: {
                    'content-encoding': 'gzip',
                },
                streamTransformers: [
                    zlib.createGzip(),
                ],
            };
        }
    }

    async interceptResponse(targetUri, existingHeaders) {
        if (targetUri.startsWith('https://my-backend-server.com') && existingHeaders
            return {
                removeHeaders: [
                    'content-encoding',
                ],
                streamTransformers: [
                    zlib.createGunzip(),
                ],
            };
        }
    }
}
```

Encrypt request/response body (only supported by the default/*streamAPI* proxy implementation):

```
import crypto from 'crypto';

export default class MyInterceptor implements MashroomHttpProxyInterceptor {

    async interceptRequest(targetUri) {
        if (targetUri.startsWith('https://my-backend-server.com')) {
            return {
                streamTransformers: [
```

```
                    crypto.createCipheriv(/* .... */),
                ],
            };
        }
    }

    async interceptResponse(targetUri, existingHeaders) {
        if (targetUri.startsWith('https://my-backend-server.com')) {
            return {
                streamTransformers: [
                    crypto.reateDecipheriv(/* .... */),
                ],
            };
        }
    }
}
```

Mashroom Add User Header Http Proxy Interceptor

If you add this plugin it will add HTTP headers with user information to all proxy backend calls. By default, it adds:

- X-USER-NAME
- X-USER-DISPLAY-NAME
- X-USER-EMAIL

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-csrf-protection** as *dependency*.

You can override the default config in your Mashroom config file like this:

```
{
  "plugins": {
        "Mashroom Http Proxy Add User Headers Interceptor": {
            "userNameHeader": "X-USER-NAME",
            "displayNameHeader": "X-USER-DISPLAY-NAME",
            "emailHeader": "X-USER-EMAIL",
            "extraDataHeaders": {},
            "targetUris": [".*"]
        }
    }
}
```

- *userNameHeader*: The HTTP header for the username (Default: X-USER-NAME)
- *displayNameHeader*: The HTTP header for the display name (Default: X-USER-DISPLAY-NAME)
- *emailHeader*: The HTTP header for the email address (Default: X-USER-EMAIL)
- *extraDataHeaders*: A mapping of *user.extraData* properties to headers (Default: {})
- *targetUris*: A list of regular expressions that match URIs that should receive the headers (Default: [.*])

Mashroom Add Access Token Http Proxy Interceptor

If you add this plugin it will add the access token from the OpenId Connect plugin to every backend call.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-csrf-protection** as *dependency*.

You can override the default config in your Mashroom config file like this:

```
{
  "plugins": {
        "Mashroom Http Proxy Add Access Token Interceptor": {
            "addBearer": false,
            "accessTokenHeader": "X-USER-ACCESS-TOKEN",
            "targetUris": [".*"]
        }
    }
}
```

- *addBearer*: Add the token as authorization bearer header (Default: true)
- *accessTokenHeader*: The HTTP header for the access token - has no effect if *addBearer* is true (Default: X-USER-ACCESS-TOKEN)
- *targetUris*: A list of regular expressions that match URIs that should receive the headers (Default: [.*])

Mashroom WebSocket

This plugin adds WebSocket support to the *Mashroom Server*. It exposes a new service that can be used to interact with clients that connect at */websocket/**.

> **NOTE**: This implementation only allows authenticated users to connect.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-websocket** as *dependency*.

And you can use the security service like this:

```
import type {MashroomWebSocketService} from '@mashroom/mashroom-websocket/type-defir

export default async (req: Request, res: Response) => {
    const webSocketService: MashroomWebSocketService = req.pluginContext.services.we

    webSocketService.addMessageListener((path) => path === '/whatever', async (messa

        // ...

        await webSocketService.sendMessage(client, {
            message: 'Hello there'
        });
    });
}
```

You can override the default config in your Mashroom config file like this:

```
{
  "plugins": {
        "Mashroom WebSocket Webapp": {
            "path": "/websocket",
            "reconnectMessageBufferFolder": null,
            "reconnectTimeoutSec": 5,
            "restrictToRoles": ["WebSocketRole"],
            "enableKeepAlive": true,
            "keepAliveIntervalSec": 15,
            "maxConnections": 2000
        }
    }
}
```

- *path*: The path where the clients can connect (Default: /websocket)

- *reconnectMessageBufferFolder*: The path where messages are temporary stored during client reconnect. When set to null or empty string, buffering is disabled. The base for relative paths is the Mashroom config file (Default: null)

- *reconnectTimeoutSec*: Time for how long are messages buffered during reconnect (Default: 5)

- *restrictToRoles*: An optional array of roles that are required to connect (Default: null)

- *enableKeepAlive*: Enable periodic keep alive messages to all clients. This is useful if you want to prevent reverse proxies to close connections because of a read timeout (Default: true)

- *keepAliveIntervalSec*: Interval for keepalive messages in seconds (Default: 15)

- *maxConnections*: Max allowed WebSocket connections per node (Default: 2000)

There will also be a **test page** available under: */websocket/test*

Reconnect to a previous session

When you connect with a client you will receive a message with your clientId from the server:

```
{
    "type": "setClientId",
    "payload": "abcdef"
}
```

When you get disconnected you should reconnect with the query parameter *?clientId=abcdef* to get all messages you missed meanwhile.

This only works if *reconnectMessageBufferFolder* is set properly.

Services

MashroomWebSocketService

The exposed service is accessible through *pluginContext.services.websocket.service*

Interface:

```
export interface MashroomWebSocketService {
    /**
     * Add a listener for message.
     * The matcher defines which messages the listener receives. The match can be ba
     * (which is the sub path where the client connected, e.g. if it connected on /w
     * or be based on the message content or both.
     */
    addMessageListener(
        matcher: MashroomWebSocketMatcher,
        listener: MashroomWebSocketMessageListener,
    ): void;

    /**
     * Remove a message listener
     */
    removeMessageListener(
        matcher: MashroomWebSocketMatcher,
        listener: MashroomWebSocketMessageListener,
    ): void;

    /**
     * Add a listener for disconnects
     */
    addDisconnectListener(listener: MashroomWebSocketDisconnectListener): void;

    /**
     * Remove a disconnect listener
     */
    removeDisconnectListener(
        listener: MashroomWebSocketDisconnectListener,
    ): void;

    /**
     * Send a (JSON) message to given client.
     */
    sendMessage(client: MashroomWebSocketClient, message: any): Promise<void>;

    /**
     * Get all clients on given connect path
     */
    getClientsOnPath(connectPath: string): Array<MashroomWebSocketClient>;

    /**
     * Get all clients for a specific username
     */
    getClientsOfUser(username: string): Array<MashroomWebSocketClient>;

    /**
     * Get the number of connected clients
```

```
     */
    getClientCount(): number;

    /**
     * Close client connection (this will also trigger disconnect listeners)
     */
    close(client: MashroomWebSocketClient): void;

    /**
     * The base path where clients can connect
     */
    readonly basePath: string;
}
```

Mashroom Messaging

This plugin adds server side messaging support to *Mashroom Server*. If an external provider plugin (e.g. MQTT) is configured the messages can also be sent across multiple nodes (cluster support!) and to 3rd party systems.

Optionally it supports sending and receiving messages via WebSocket (Requires *mashroom-websocket*).

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-messaging** as *dependency*.

And you can use the messaging service like this:

```
import type {MashroomMessagingService} from '@mashroom/mashroom-messaging/type-defir

export default async (req: Request, res: Response) => {
    const messagingService: MashroomMessagingService = req.pluginContext.services.me

    // Subscribe
    await messagingService.subscribe(req, 'my/topic', (data) => {
        // Do something with data
    });

    // Publish
    await messagingService.publish(req, 'other/topic', {
        item: 'Beer',
        quantity: 1,
    });

    // ...
}
```

You can override the default config in your Mashroom config file like this:

```
{
  "plugins": {
        "Mashroom Messaging Services": {
            "externalProvider": null,
            "externalTopics": [],
            "userPrivateBaseTopic": "user",
            "enableWebSockets": true,
            "topicACL": "./topicACL.json"
        }
    }
}
```

- *externalProvider*: A plugin that connects to an external messaging system. Allows to receive messages from other systems and to send messages "out" (Default: null)
- *externalTopics*: A list of topic roots that should be considered as external. E.g. if the list contains *other-system* topics published to *other-system/foo* or *other-system/bar* would be sent via *externalProvider*. (Default: [])
- *userPrivateBaseTopic*: The base for private user topics. If the prefix is *something/user* the user *john* would only be able to subscribe to *user/john/something* and not to *something/user/thomas/weather-update* (Default: user).
- *enableWebSockets*: Enable WebSocket support when *mashroom-websocket* is present (Default: true)
- *topicACL*: Access control list to restrict the use of certain topic patterns to specific roles (Default: ./topicACL.json)

With a config like that you can place a file *topic*acl.json_ in your server config with a content like this:

```
{
    "$schema": "https://www.mashroom-server.com/schemas/mashroom-security-topic-acl.
    "my/topic": {
        "allow": ["Role1"]
    },
    "foo/bar/#": {
        "allow": "any"
        "deny": ["NotSoTrustedRole"]
    }
}
```

The general structure is:

```
    "/my/+/topic/#": {
        "allow": "any"|<array of roles>
        "deny": "any"|<array of roles>
    }
```

You can use here + or * as a wildcard for a single level and # for multiple levels.

WebSocket interface

If *enableWebSockets* is true you can connect to the messaging system on */messaging* which is by default **/websocket/messaging**. The server expects and sends serialized JSON.

After a successful connection you can use the following commands:

Subscribe

```
{
  messageId: 'ABCD',
  command: 'subscribe',
  topic: 'foo/bar',
}
```

The messageId should be unique. You will get a response message like this when the operation succeeds:

```
{
  messageId: 'ABCD',
  success: true,
}
```

Otherwise a error message like this:

```
{
  messageId: 'ABCD',
  error: true,
  message: 'The error message'
}
```

Unsubscribe

```
{
  messageId: 'ABCD',
  command: 'unsubscribe',
```

```
  topic: 'foo/bar',
}
```

Success and error response messages are the same as above.

Publish

```
{
  messageId: 'ABCD',
  command: 'publish',
  topic: 'foo/bar',
  message: {
     foo: 'bar'
  }
}
```

Success and error response messages are the same as above.

And the server will push the following **if a message for a subscribed topic arrives**:

```
{
  remoteMessage: true,
  topic: 'foo/bar',
  message: {
     what: 'ever'
  }
}
```

Services

MashroomMessagingService

The exposed service is accessible through *pluginContext.services.messaging.service*

Interface:

```
export interface MashroomMessagingService {
    /**
     * Subscribe to given topic.
     * Topics can be hierarchical and also can contain wildcards. Supported wildcard
     * and # for multiple levels. E.g. foo/+/bar or foo/#
     *
     * Throws an exception if there is no authenticated user
     */
    subscribe(req: Request, topic: string, callback: MashroomMessagingSubscriberCall

    /**
     * Unsubscribe from topic
```

```
     */
    unsubscribe(topic: string, callback: MashroomMessagingSubscriberCallback): Promi

    /**
     * Publish to a specific topic
     *
     * Throws an exception if there is no authenticated user
     */
    publish(req: Request, topic: string, data: any): Promise<void>;

    /**
     * The private topic only the current user can access.
     * E.g. if the value is user/john the user john can access to user/john/whatever
     * but not to user/otheruser/foo
     *
     * Throws an exception if there is no authenticated user
     */
    getUserPrivateTopic(req: Request): string;

    /**
     * The connect path to send publish or subscribe via WebSocket.
     * Only available if enableWebSockets is true and mashroom-websocket is preset.
     */
    getWebSocketConnectPath(req: Request): string | null | undefined;
}
```

Plugin Types

external-messaging-provider

This plugin type connects the messaging system to an external message broker. It also adds cluster support to the messaging system.

To register your custom external-messaging-provider plugin add this to *package.json*:

```
{
    "mashroom": {
        "plugins": [
            {
                "name": "My Custom External Messaging Provider",
                "type": "external-messaging-provider",
                "bootstrap": "./dist/mashroom-bootstrap",
                "defaultConfig": {
                   "myProperty": "foo"
                }
            }
        ]
    }
}
```

The bootstrap returns the provider:

```
import type {MashroomExternalMessagingProviderPluginBootstrapFunction} from '@mashrc

const bootstrap: MashroomExternalMessagingProviderPluginBootstrapFunction = async (p

    return new MyExternalMessagingProvider(/* ... */);
};

export default bootstrap;
```

The provider has to implement the following interface:

```
export interface MashroomMessagingExternalProvider {
    /**
     * Add a message listener
     * The message must be a JSON object.
     */
    addMessageListener(listener: MashroomExternalMessageListener): void;

    /**
     * Remove an existing listener
     */
    removeMessageListener(listener: MashroomExternalMessageListener): void;

    /**
     * Send a message to given internal topic.
     * Used to broadcast message between Mashroom instances.
     *
     * The passed topic must be prefixed with the topic the provider is listening to
     * E.g. if the passed topic is foo/bar and the provider is listening to mashroom
     * sent to mashroom/foo/bars.
     *
     * The message will be a JSON object.
     */
    sendInternalMessage(topic: string, message: any): Promise<void>;

    /**
     * Send a message to given external topic.
     * Used to send messages to 3rd party systems.
     *
     * The message will be a JSON object.
     */
    sendExternalMessage(topic: string, message: any): Promise<void>;
}
```

Mashroom Messaging External Provider AMQP

This plugin allows to use an AMQP 1.0 compliant broker as external messaging provider for server side messaging. This enables cluster support for server side messaging and also allows communication with 3rd party systems.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-messaging-external-provider-amqp** as *dependency*.

To activate this provider configure the *Mashroom Messaging* plugin like this:

```
{
    "plugins": {
        "Mashroom Messaging Services": {
            "externalProvider": "Mashroom Messaging External Provider AMQP"
        }
    }
}
```

And configure this plugin like this in the Mashroom config file:

```
{
  "plugins": {
        "Mashroom Messaging External Provider MQTT": {
              "internalRoutingKey": "mashroom",
              "brokerTopicExchangePrefix": "/topic/",
              "brokerTopicMatchAny": "#",
              "brokerHost": "localhost",
              "brokerPort": 5672,
              "brokerUsername": null,
              "brokerPassword": null
        }
    }
}
```

- *internalRoutingKey*: The base routing key the server should use for internal messages. E.g. if the value is *mashroom.test* all messages published internally are prefixed with *mashroom.test* before published to the broker and at the same time this provider listens to *mashroom.test.#* for messages (Default: mashroom)

- *brokerTopicExchangePrefix*: The prefix for the topic exchange (default: /topic/ (RabbitMQ))

- *brokerTopicMatchAny*: The wildcard for match any words (default: # (RabbitMQ))

- *brokerHost*: AMQP broker host (Default: localhost)
- *brokerPort*: AMQP broker port (Default: 5672)
- *brokerUsername*: AMQP broker username (optional)
- *brokerPassword*: AMQP broker password (optional)

Broker specific configuration

RabbitMQ

```
"brokerTopicExchangePrefix": "/topic/",
"brokerTopicMatchAny": "#",
```

ActiveMQ

```
"brokerTopicExchangePrefix": "topic://",
"brokerTopicMatchAny": ">",
```

Qpid Broker

```
"brokerTopicExchangePrefix": "amq.topic/",
"brokerTopicMatchAny": "#",
```

Mashroom Messaging External Provider MQTT

This plugin allows to use a MQTT server as external messaging provider for server side messaging. This enables cluster support for server side messaging and also allows communication with 3rd party systems.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-messaging-external-provider-mqtt** as *dependency*.

To activate this provider configure the *Mashroom Messaging* plugin like this:

```
{
    "plugins": {
        "Mashroom Messaging Services": {
            "externalProvider": "Mashroom Messaging External Provider MQTT"
        }
```

```
    }
}
```

And configure this plugin like this in the Mashroom config file:

```
{
  "plugins": {
        "Mashroom Messaging External Provider MQTT": {
            "internalTopic": "mashroom",
            "mqttConnectUrl": "mqtt://localhost:1883",
            "mqttProtocolVersion": 4,
            "mqttQoS": 1,
            "mqttUser": null,
            "mqttPassword": null,
            "rejectUnauthorized": true
        }
    }
}
```

- *internalTopic*: The base topic the server should use for internal messages. E.g. if the value is *mashroom/test* all messages published internally are prefixed with *mashroom/test* before published to MQTT and at the same time this provider listens to *mashroom/test/#* for messages (Default: mashroom)
- *mqttConnectUrl*: MQTT connect URL (Default: mqtt://localhost:1883)
- *mqttProtocolVersion*: MQTT protocol version (3, 4 or 5) (Default: 4)
- *mqttQoS*: Quality of service level (0, 1, or 2) (Default: 1)
- *mqttUser*: Optional MQTT username (Default: null)
- *mqttPassword*: Optional MQTT password (Default: null)
- *rejectUnauthorized*: If you use mqtts or wss with a self-signed certificate set it to false (Default: true)

Mashroom Messaging External Provider Redis

This plugin allows to use a Redis server as external messaging provider for server side messaging. This enables cluster support for server side messaging and also allows communication with 3rd party systems.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-messaging-external-provider-redis** as *dependency*.

To activate this provider configure the *Mashroom Messaging* plugin like this:

```
{
    "plugins": {
        "Mashroom Messaging Services": {
            "externalProvider": "Mashroom Messaging External Provider Redis"
        }
    }
}
```

And configure this plugin like this in the Mashroom config file:

```
{
  "plugins": {
        "Mashroom Messaging External Provider Redjs": {
            "internalTopic": "mashroom",
            "client": {
                "redisOptions": {
                    "host": "localhost",
                    "port": "6379",
                    "maxRetriesPerRequest": 3,
                    "enableOfflineQueue": false
                },
                "cluster": false,
                "clusterNodes": null,
                "clusterOptions": null
            }
        }
    }
}
```

- *internalTopic*: The base topic the server should use for internal messages. E.g. if the value is *mashroom/test* all messages published internally are prefixed with *mashroom/test* before published to MQTT and at the same time this provider listens to *mashroom/test/#* for messages (Default: mashroom)

- *client*: Options for the Redis client. *redisOptions* are just to the *Redis* constructor of [ioredis](https://github.com/luin/ioredis Checkout out the ioredis documentation for all available options.

Mashroom Memory Cache

This plugin adds a general purpose memory cache service. Some other plugins will automatically use it if present, for example *mashroom-storage*.

The cache service provides multiple *regions* with the possibility to clear single regions. It comes with a built-in provider that uses the local Node.js memory, which is not ideal for clusters. But it can also be configured to use another provider, e.g. an implementation based on *Redis*.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-memory-cache** as *dependency*.

You can override the default config in your Mashroom config file like this:

```
{
    "plugins": {
        "Mashroom Memory Cache Services": {
            "provider": "local",
            "defaultTTLSec": 300
        }
    }
}
```

- *provider*: The name of the provider. Default is *local* which uses the local Node.js memory.
- *defaultTTLSec*: The default TTL in seconds (Default: 300)

Services

MashroomMemoryCacheService

The exposed service is accessible through *pluginContext.services.cache.service*

Interface:

```
export interface MashroomMemoryCacheService {
    /**
     * Get a cache entry from given region
     */
    get(region: string, key: CacheKey): Promise<CacheValue | undefined>;
    /**
     * Set a cache entry in given region
     */
    set(region: string, key: CacheKey, value: CacheValue, ttlSec?: number): Promise<
    /**
     * Delete an entry in given region
     */
    del(region: string, key: CacheKey): Promise<void>;
    /**
     * Clear the entire region
```

```
     * This might be an expensive operation, depending on the provider
     */
    clear(region: string): Promise<void>;
    /**
     * Get the number of entries in this region (if possible)
     * This might be an expensive operation, depending on the provider
     */
    getEntryCount(region: string): Promise<number | undefined>;
}
```

Plugin Types

memory-cache-provider

This plugin type adds a memory cache provider that can be used by this plugin.

To register a custom memory-cache-provider plugin add this to *package.json*:

```
{
    "mashroom": {
        "plugins": [
            {
                "name": "My Cache Store Provider",
                "type": "memory-cache-provider",
                "bootstrap": "./dist/mashroom-bootstrap.js",
                "defaultConfig": {
                    "myProperty": "test"
                }
            }
        ]
    }
}
```

The bootstrap returns the provider:

```
import type {MashroomMemoryCacheProviderPluginBootstrapFunction} from '@mashroom/mas

const bootstrap: MashroomMemoryCacheProviderPluginBootstrapFunction = async (pluginN
    return new MyCacheStore();
};

export default bootstrap;
```

And the provider has to implement the following interface:

```
export interface MashroomMemoryCacheProvider {
    /**
```

```
     * Get a cache entry from given region
     */
    get(region: string, key: CacheKey): Promise<CacheValue | undefined>;
    /**
     * Set a cache entry in given region
     */
    set(region: string, key: CacheKey, value: CacheValue, ttlSec: number): Promise<v
    /**
     * Delete an entry in given region
     */
    del(region: string, key: CacheKey): Promise<void>;
    /**
     * Clear the entire region
     */
    clear(region: string): Promise<void>;
    /**
     * Get the number of entries in this region (if possible)
     */
    getEntryCount(region: string): Promise<number | undefined>;
}
```

Mashroom Memory Cache Redis Provider

This plugin adds a *Redis* based provider for the *mashroom-memory-cache*.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-memory-cache-provider-redis** as *dependency*.

To activate this provider configure the *Mashroom Memory Cache* plugin like this:

```
{
    "plugins": {
          "Mashroom Memory Cache Redis Provider": {
              "provider": "Mashroom Memory Cache Redis Provider",
              "defaultTTLSec": 10
          }
    }
}
```

And configure this plugin like this in the Mashroom config file:

```
{
  "plugins": {
        "Mashroom Memory Cache Redis Provider": {
            "redisOptions": {
                "host": "localhost",
                "port": "6379",
```

```
                "keyPrefix": "mashroom:cache:"
            }
        }
    }
}
```

- *redisOptions*: Passed to the *Redis* constructor of ioredis

Checkout out the ioredis documentation for all available options.

Usage with Sentinel

For a high availability cluster with Sentinel the configuration would look like this:

```
{
  "plugins": {
      "Mashroom Memory Cache Redis Provider": {
          "redisOptions": {
              "sentinels": [
                  { "host": "localhost", "port": 26379 },
                  { "host": "localhost", "port": 26380 }
               ],
              "name": "myMaster",
              "keyPrefix": "mashroom:cache:"
          }
      }
  }
}
```

- *sentinels*: List of sentinel nodes to connect to
- *name*: Identifies a group of Redis instances composed of a master and one or more slaves

Checkout out the *Sentinel* section of the ioredis documentation for all available options.

Usage with a cluster

For a sharding cluster configure the plugin like this:

```
{
  "plugins": {
      "Mashroom Memory Cache Redis Provider": {
          "cluster": true,
          "clusterNodes": [
```

```
            {
                "host": "redis-node1",
                "port": "6379"
            },
            {
                "host": "redis-node2",
                "port": "6379"
            }
        ],
        "clusterOptions": {
            "maxRedirections": 3
        },
        "redisOptions": {
            "keyPrefix": "mashroom:cache:"
        }
    }
  }
}
```

- *cluster*: Enables cluster support, must be true
- *clusterNodes*: Cluster start nodes
- *clusterOptions*: Passed as second argument of the *Redis.Cluster* constructor of *ioredis*
- *redisOptions*: Passed as *redisOptions* in the *clusterOptions*

Checkout out the *Cluster* section of the ioredis documentation for all available options.

Mashroom I18N

This plugin adds a service for internationalization. It determines the language from the HTTP headers and supports translation of messages.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-i18n** as *dependency*.

After that you can use the service like this:

```
import type {MashroomI18NService} from '@mashroom/mashroom-i18n/type-definitions';

export default (req: Request, res: Response) => {
    const i18nService: MashroomI18NService = req.pluginContext.services.i18n.service

    const currentLang = i18nService.getLanguage(req);
```

```
    const message =  i18nService.getMessage('username', 'de');
    // message will be 'Benutzernamen'

    // ...
}
```

You can override the default config in your Mashroom config file like this:

```
{
    "plugins": {
        "Mashroom Internationalization Services": {
            "availableLanguages": ["en", "de", "fr"],
            "defaultLanguage": "en",
            "messages": "./messages"
        }
    }
}
```

- *availableLanguages*: A list of available languages (Default: ["en"])
- *defaultLanguage*: The default language if it can not be determined from the request (Default: en)
- *messages*: The folder with custom i18n messages (Default: ./messages). There are default messages in the *messages* folder of this package.

The lookup for message files works like this:

- <messages_folder>/messages.<lang>.json
- <built*in*messages_folder>/messages.<lang>.json
- <messages_folder>/messages.json
- <built*in*messages_folder>/messages.json

And a messages file (e.g. *messages.de.json*) looks like this:

```
{
    "message_key": "Die Nachricht"
}
```

Services

The exposed service is accessible through *pluginContext.services.i18n.service*

Interface:

```
export interface MashroomI18NService {
    /**
     * Get the currently set language (for current session)
     */
    getLanguage(req: Request): string;

    /**
     * Set session language
     */
    setLanguage(language: string, req: Request): void;

    /**
     * Get the message for given key and language
     */
    getMessage(key: string, language: string): string;

    /**
     * Get plain string in the current users language from a I18NString
     */
    translate(req: Request, str: I18NString): string;

    /**
     * Get available languages
     */
    readonly availableLanguages: Readonly<Array<string>>;

    /**
     * Get the default languages
     */
    readonly defaultLanguage: string;
}
```

Mashroom Background Jobs

This plugin adds a background job scheduler to the *Mashroom Server* that supports cron expressions. It is possible to add background jobs via service or as custom plugin.

If you don't provide a cron expression the job is only executed now (immediately), this is useful if you need some code that should be executed once during server startup.

This plugin also comes with an Admin UI extension (*/mashroom/admin/ext/background-jobs*) that can be used to check the jobs.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-background-jobs** as *dependency*.

After that you can use the service like this:

```
import type {MashroomBackgroundJobService} from '@mashroom/mashroom-background-jobs/

export default async (req: Request, res: Response) => {
    const backgroundJobsService: MashroomBackgroundJobService = req.pluginContext.se

    backgroundJobsService.schedule('Test Job', '0/5 * * * *', () => {
       // Job implementation
    });

    // ...
}
```

NOTE: Despite its name a job is started in the main thread and therefore blocks the event loop. So, if you do CPU intensive work you need to spawn a Worker Thread yourself.

Services

MashroomBackgroundJobService

The exposed service is accessible through *pluginContext.services.backgroundJobs.service*

Interface:

```
export interface MashroomBackgroundJobService {

    /**
     * Schedule a job.
     * If cronSchedule is not defined the job is executed once (immediately).
     * Throws an error if the cron expression is invalid.
     */
    scheduleJob(name: string, cronSchedule: string | undefined | null, callback: Mas

    /**
     * Unschedule an existing job
     */
    unscheduleJob(name: string): void;

    readonly jobs: Readonly<Array<MashroomBackgroundJob>>;
}
```

Plugin Types

This plugin type allows it to schedule a background job.

To register your custom background-job plugin add this to *package.json*:

```
{
    "mashroom": {
        "plugins": [
            {
                "name": "My background job",
                "type": "background-job",
                "bootstrap": "./dist/mashroom-bootstrap.js",
                "defaultConfig": {
                   "cronSchedule": "0/1 * * * *",
                   "invokeImmediately": false,
                   "yourConfigProp": "whatever"
                }
            }
        ]
    }
}
```

- *cronSchedule*: The execution schedule for the job, must be a valid cron expression, see node-cron ; if this is null or undefined the job is executed exactly one during startup.
- *invokeImmediately*: Optional hint that the job should additionally be invoked immediately (Default: false)

The bootstrap returns the job callback:

```
import type {MashroomBackgroundJobPluginBootstrapFunction} from '@mashroom/mashroom-

const bootstrap: MashroomBackgroundJobPluginBootstrapFunction = async (pluginName, p
    const {yourConfigProp} = pluginConfig;
    return (pluginContect) => {
        // Job impl
    };
};

export default bootstrap;
```

Mashroom Browser Cache

This plugin adds a Service to manage cache control headers. It also allows to disable the cache globally.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-browser-cache** as *dependency*.

After that you can use the service like this:

```
import type {MashroomCacheControlService} from '@mashroom/mashroom-browser-cache/typ

export default async (req: Request, res: Response) => {

    const cacheControlService: MashroomCacheControlService = req.pluginContext.servi
    await cacheControlService.addCacheControlHeader('ONLY_FOR_ANONYMOUS_USERS', req,

    // ..
};
```

You can override the default config in your Mashroom config file like this:

```
{
  "plugins": {
        "Mashroom Cache Control Services": {
            "disabled": false,
            "maxAgeSec": 31536000
        }
    }
}
```

- *disabled*: Disable browser caching (default: false)
- *maxAgeSec*: Max age in seconds (default: 31536000 (30d))

Services

MashroomCacheControlService

The Cache Control service is accessible through *pluginContext.services.browserCache.cacheControl*

Interface:

```
export interface MashroomCacheControlService {
    /**
     * Add the Cache-Control header based on the policy and authentication status.
     */
     addCacheControlHeader(cachingPolicy: CachingPolicy, request: Request, response:
```

```
    /**
     * Remove a previously set Cache-Control header
     */
     removeCacheControlHeader(response: Response): void;
}
```

Caching Policies are:

```
export type CachingPolicy =  'SHARED' | 'PRIVATE_IF_AUTHENTICATED' | 'NEVER' | 'ONLY
```

Mashroom CDN

This plugin adds a Service to manage CDN hosts. It basically just returns a host from a configurable list, which can be used to access an asset via CDN.

> **NOTE**: The *mashroom-cdn* plugin requires a CDN that works like a transparent proxy, which forwards an identical request to the *origin* (in this case Mashroom) if does not exist yet.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-cdn** as *dependency*.

After that you can use the service like this:

```
import type {MashroomCDNService} from '@mashroom/mashroom-cdn/type-definitions';

export default async (req: Request, res: Response) => {

    const cdnService: MashroomCDNService = req.pluginContext.services.cdn.service;

    const cdnHost = cdnService.getCDNHost();
    const resourceUrl = `${cdnHost}/<the-actual-path>`;

    // ..
};
```

You can override the default config in your Mashroom config file like this:

```
{
  "plugins": {
        "Mashroom CDN Services": {
            "cdnHosts": [
                "//cdn1.myhost.com",
                "//cdn2.myhost.com"
```

```
            ]
        }
    }
}
```

- *cdnHosts*: A list of CDN hosts (default: [])

Services

MashroomCDNService

The CDN service is accessible through *pluginContext.services.cdn.cacheControl*

Interface:

```
export interface MashroomCDNService {
    /**
     * Return a CDN host or null if there is none configured.
     */
    getCDNHost(): string | null;
}
```

Mashroom Robots

This plugin adds a middleware that exposes a robots.txt file for search engines.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-robots** as *dependency*.

You can override the default config in your Mashroom config file like this:

```
{
    "plugins": {
        "Mashroom Robots Middleware": {
            "robots.txt": "./robots.txt"
        }
    }
}
```

- *robots.txt*: The path to the robots.txt file. Can be relative to the server config file or absolute. If not provided, the default config denies the access to all search engines.

Mashroom Virtual Host Path Mapper

This plugin adds the possibility to map external paths to internal ones based on virtual host. This is required for web-apps that need to know the actual "base path" to generate URLs (in that case rewriting via reverse proxy won't work).

For example *Mashroom Portal* can use this to move *Sites* to different paths but keep the ability to generate absolute paths for resources and API calls. Which is useful if you want to expose specific *Sites* via a virtual host.

> **NOTE**: All other plugins will only deal with the rewritten paths, keep that in mind especially when defining ACLs.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-vhost-path-mapper** as *dependency*.

To map for example a portal site to *www.my-company.com/web* configure the reverse proxy like this:

```
www.my-company.com/web -> <portal-host>:<portal-port>/
```

and the plugin like this:

```
{
  "plugins": {
       "Mashroom VHost Path Mapper Middleware": {
           "considerHttpHeaders": ["x-my-custom-host-header", "x-forwarded-host"],
           "hosts": {
              "www.my-company.com": {
                 "frontendBasePath": "/web",
                 "mapping": {
                    "/login": "/login",
                    "/": "/portal/public-site"
                 }
              },
              "localhost:8080": {
                 "mapping": {
                     "/": "/local-test"
                 }
              }
          }
       }
    }
}
```

That means if someone accesses *Mashroom Server* via *https://www.my-company.com/web/test* the request will hit the path */portal/public-site/test*.

It also works the other way round. If the server redirects to */login* it would be changed to */web/login* (in this example).

Port based virtual hosts (like localhost:8080) are also possible but only if the request still contains the original *host* header (and no *X-Forwarded-Host* different from the *host* header).

The mapping rules do not support regular expressions.

The *frontendBasePath* is optional and / by default.

The *considerHttpHeaders* property is also optional and can be used to detect the host based on some custom header. The first header that is present will be used (so the order in the list specifies the priority).

Services

MashroomVHostPathMapperService

The exposed service is accessible through *pluginContext.services.vhostPathMapper.service*

Interface:

```
export interface MashroomVHostPathMapperService {
    /**
     * Reverse map the given server url to the url as seen by the user (browser).
     * The given URL must not contain host, only path with query params and so on.
     */
    getFrontendUrl(req: Request, url: string): string;

    /**
     * Get the details if the url of the current path has been rewritten
     */
    getMappingInfo(req: Request): RequestVHostMappingInfo | undefined;
}
```

Mashroom Monitoring Metrics Collector

This plugin provides a service to add metrics to the monitoring system that can be used by plugins. It also adds a middleware that collects request metrics like duration and HTTP status.

It uses internally the OpenTelemetry SDK .

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-monitoring-stats-collector** as *dependency*.

You can change the default configuration in your Mashroom config file like this:

```
{
  "plugins": {
        "Mashroom Monitoring Metrics Collector Services": {
            "disableMetrics": [],
            "defaultHistogramBuckets": [0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1,
            "customHistogramBucketConfig": {
              "mashroom_http_request_duration_seconds": [0.1, 1, 10]
            }
        }
    }
}
```

- *disableMetrics*: A list of metrics that should be disabled
- *defaultHistogramBuckets*: Default buckets for histogram metrics
- *customHistogramBucketConfig*: Override the bucket configuration for histogram metrics

Synchronous example:

```
    const collectorService: MashroomMonitoringMetricsCollectorService = req.pluginCo

    collectorService.counter('http_request_counter', 'HTTP Request Counter').inc();
```

Asynchronous example:

```
    const collectorService: MashroomMonitoringMetricsCollectorService = pluginContex

    const ref = await collectorService.addObservableCallback((asyncCollectorService)
        // ... somehow get the value to measure
        asyncCollectorService.gauge('http_pool_active_connections', 'HTTP Pool Activ
    });
```

Using directly the OpenTelemetry API

If you prefer the OpenTelemetry API the above examples would look like:

```
    const collectorService: MashroomMonitoringMetricsCollectorService = req.pluginCo

    const meterProvider = collectorService.getOpenTelemetryMeterProvider();
    const meter = meterProvider.getMeter('my_meter');
```

```
    const counter = meter.createCounter('http_request_counter', {
        description: 'HTTP Request Counter',
    });

    counter.add(1);
```

and this:

```
    const collectorService: MashroomMonitoringMetricsCollectorService = pluginContex

    const meterProvider = collectorService.getOpenTelemetryMeterProvider();
    const meter = meterProvider.getMeter('my_meter');
    const observableGauge = meter.createObservableGauge('http_request_counter', {
        description: 'HTTP Request Counter',
    });

    meter.addBatchObservableCallback((observableResult) => {
        // ... somehow get the value to measure
        observableResult.observe(observableGauge, theValue);
    }, [observableGauge]);
```

Services

MashroomMonitoringMetricsCollectorService

The exposed service is accessible through *pluginContext.services.metrics.service*

Interface:

```
/**
 * Mashroom Monitoring Metrics Collector Service
 *
 * An abstraction that uses currently OpenTelemetry Metrics under the hood, see http
 */
export interface MashroomMonitoringMetricsCollectorService {
    /**
     * A counter is a cumulative metric that represents a single monotonically incre
     * whose value can only increase.
     * Even though the returned Counter has a set() method, the new value must alway
     */
    counter(name: string, help: string): MashroomMonitoringMetricsCounter;
    /**
     * A histogram samples observations (usually things like request durations or re
     * and counts them in configurable buckets.It also provides a sum of all observe
     */
    histogram(name: string, help: string, buckets?: number[]): MashroomMonitoringMet
    /**
     * Add a callback for asynchronous measuring of values.
```

```
     * Gauges and counters where you can set the value directly are only available l
     */
    addObservableCallback(cb: MashroomMonitoringMetricsObservableCallback): Promise<
    /**
     * Get OpenTelemetry resource metrics for export
     */
    getOpenTelemetryResourceMetrics(): Promise<ResourceMetrics>;
    /**
     * The underlying MeterProvider which can be used if you prefer directly using t
     * All metrics created will be automatically exported as well.
     */
    getOpenTelemetryMeterProvider(): MeterProvider;
}
```

NOTE: Don't keep a global reference to the returned metric objects.

Mashroom Monitoring Prometheus Exporter

This plugin exports the following metrics to the Prometheus monitoring system:

- The standard metrics like CPU and memory usage described here
- Additional V8 GC metrics (if the module *prometheus-gc-stats* is present)
- Loaded plugins with status
- HTTP request durations and response codes
- Mashroom plugin metrics like session count, http proxy stats, memory cache stats, MongoDB/Redis connection stats, …

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-monitoring-prometheus-exporter** as *dependency*.

You can change the default configuration in your Mashroom config file like this:

```
{
  "plugins": {
        "Mashroom Monitoring Prometheus Exporter Webapp": {
            "path": "/myMetricsPath"
        }
    }
}
```

- *path*: The path where the metrics will be exported (Default: /metrics)

Example Queries

> The examples assume the Prometheus job to scrape the metrics adds the label service:Mashroom

Request rate:

```
sum(rate(mashroom_http_requests_total{service="Mashroom"}[5m]))
```

Requests with HTTP 500 response rate:

```
sum(rate(mashroom_http_requests_total{status="500", service="Mashroom"}[5m]))
```

95% of requests served within seconds:

```
histogram_quantile(0.95, sum(rate(mashroom_http_request_duration_seconds_bucket{serv
```

Heap total in MB:

```
nodejs_heap_size_total_bytes{service="Mashroom"} / 1024 / 1024
```

Heap used in MB:

```
nodejs_heap_size_used_bytes{service="Mashroom"} / 1024 / 1024
```

CPU usage total in %:

```
avg(irate(process_cpu_seconds_total{service="Mashroom"}[5m])) * 100
```

GC pauses 95% quantile

```
histogram_quantile(0.95, sum(rate(nodejs_gc_duration_seconds_bucket[5m])) by (le))
```

User sessions:

```
mashroom_sessions_total{service="Mashroom"}
```

Active HTTP proxy connections (e.g. Portal App REST calls):

```
mashroom_http_proxy_http_pool_connections_active_total{service="Mashroom"}
mashroom_http_proxy_https_pool_connections_active_total{service="Mashroom"}
mashroom_http_proxy_ws_connections_active_total{service="Mashroom"}
```

Idle HTTP proxy connections:

```
mashroom_http_proxy_http_pool_connections_idle_total{service="Mashroom"}
mashroom_http_proxy_https_pool_connections_idle_total{service="Mashroom"}
```

Plugins total:

```
mashroom_plugins_total{service="Mashroom"}
```

Plugins loaded:

```
mashroom_plugins_loaded_total{service="Mashroom"}
```

Plugins in error state:

```
mashroom_plugins_error_total{service="Mashroom"}
```

Remote Portal App endpoints total:

```
mashroom_remote_app_endpoints_total{service="Mashroom"}
```

Remote Portal App endpoints in error state:

```
mashroom_remote_app_endpoints_error_total{service="Mashroom"}
```

Kubernetes remote Portal App services total:

```
mashroom_remote_app_k8s_services_total{service="Mashroom"}
```

Kubernetes remote Portal App services in error state:

```
mashroom_remote_app_k8s_services_error_total{service="Mashroom"}
```

Memory cache hit ratio:

```
mashroom_memory_cache_hit_ratio{service="Mashroom"}
```

Redis session store provider connected:

```
mashroom_sessions_redis_nodes_connected{service="Mashroom"}
```

Redis memory cache provider connected:

```
mashroom_memory_cache_redis_nodes_connected{service="Mashroom"}
```

MongoDB storage provider connected:

```
mashroom_storage_mongodb_connected{service="Mashroom"}
```

MQTT messaging system connected:

```
mashroom_messaging_mqtt_connected{service="Mashroom"}
```

Kubernetes Hints

On Kubernetes the metrics are scraped separately for each container. So, you have to do the aggregation in the query.

For example, the overall request rate would still be:

```
sum(rate(mashroom_http_requests_total{namespace="my-namespace"}[5m]))
```

But the request rate per pod:

```
sum by (kubernetes_pod_name) (rate(mashroom_http_requests_total{namespace="my-namesp
```

Or the Session count per pod:

```
mashroom_sessions_total{namespace="my-namespace"} by (kubernetes_pod_name)
```

In the last two examples you typically would use {{kubernetes*pod*name}} in the legend.

Demo Grafana Dashboard

You can find a demo Grafana Dashboard here:
https://github.com/nonblocking/mashroom/tree/master/packages/plugin-packages/mashroom-

monitoring-prometheus-exporter/test/grafana-test/grafana/provisioning/dashboards/Mashroom%20Dashboard.json

Mashroom Monitoring PM2 Exporter

This plugin exports metrics to the PM2 via pm2/io . Which is useful if you use the PM2 process manager to run *Mashroom Server*.

It activates the pm2/io default metrics like v8, runtime, network, http (configurable). And it exports *Mashroom* plugin metrics like session count, memory cache stats, MongoDB/Redis connection stats, …

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-monitoring-pm2-exporter** as *dependency*.

You can change the default configuration in your Mashroom config file like this:

```
    "plugins": {
        "Mashroom Monitoring PM2 Exporter": {
            "pmxMetrics": {
                "v8": true,
                "runtime": true,
                "network": {
                    "upload": true,
                    "download": true
                },
                "http": true,
                "eventLoop": true
            },
            "mashroomMetrics": [
                "mashroom_plugins_total",
                "mashroom_plugins_loaded_total",
                "mashroom_plugins_error_total",
                "mashroom_remote_app_endpoints_total",
                "mashroom_remote_app_endpoints_error_total",
                "mashroom_sessions_total",
                "mashroom_websocket_connections_total",
                "mashroom_https_proxy_active_connections_total",
                "mashroom_https_proxy_idle_connections_total",
                "mashroom_https_proxy_waiting_requests_total",
                "mashroom_sessions_mongodb_connected",
                "mashroom_sessions_redis_nodes_connected",
                "mashroom_storage_mongodb_connected",
                "mashroom_memory_cache_entries_added_total",
                "mashroom_memory_cache_hit_ratio",
                "mashroom_memory_cache_redis_nodes_connected",
                "mashroom_messaging_amqp_connected",
                "mashroom_messaging_mqtt_connected"
            ]
```

```
        }
    }
}
```

- *pmxMetrics*: Will be passed as *metrics* to the pm2/io configuration
- *mashroomMetrics*: A list of Mashroom plugin metrics that should be exposed.

NOTE: Currently only *counter* and *gauge* metrics can be exported! For a full list install the *mashroom-monitoring-prometheus-exporter* and check the output of /metrics

After starting the server with pm2 you can see the metrics in the "Custom metrics" pane when you start:

```
pm2 monit
```

Or you can get it as JSON (*axm_monitor* property) if you execute

```
pm2 prettylist
```

Fetching all metrics via inter-process communication

If you want to gather all metrics in OpenTelemetry format you can use inter-process communication.

Here as an example how to export the metrics for each worker and make it available in Prometheus format:

Create a script metrics.js with a simple webserver:

```
const pm2 = require('pm2');
const { PrometheusSerializer } = require('@opentelemetry/exporter-prometheus');
const Express = require('express');
const metricsServer = Express();

const metrics = {}; // <pid> -> OpenTelemetry ResourceMetrics
const metricsServerPort = 15050;
const prometheusSerializer = new PrometheusSerializer();

metricsServer.get('/metrics/:id', async (req, res) => {
    const id = req.params.id;
    const slice = metrics[id];
    if (!slice) {
        console.error(`No metrics found for ID ${id}. Known node IDs:`, Object.keys(
```

```
        res.sendStatus(404);
        return;
    }
    res.set('Content-Type', 'text/plain');
    res.end(prometheusSerializer.serialize(slice));
});

metricsServer.listen(metricsServerPort, '0.0.0.0', () => {
    console.debug(`Prometheus cluster metrics are available at http://localhost:${me
});

setInterval(() => {
    pm2.connect(() => {
        pm2.describe('mashroom', (describeError, processInfo) => {
            if (!describeError) {
                Promise.all(processInfo.map((processData) => {
                    console.debug(`Asking process ${processData.pm_id} for metrics`)
                    return new Promise((resolve) => {
                        pm2.sendDataToProcessId(
                            processData.pm_id,
                            {
                                data: null,
                                topic: 'getMetrics',
                                from: process.env.pm_id,
                            },
                            (err, res) => {
                                if (err) {
                                    console.error('Error sending data via PM2 interc
                                }
                                resolve();
                            },
                        );
                    });
                })).finally(() => {
                    pm2.disconnect();
                });
            } else {
                pm2.disconnect();
            }
        });
    });
}, 10000);

process.on('message', (msg) => {
    if (msg.from !== process.env.pm_id && msg.topic === 'returnMetrics') {
        console.debug(`Received metrics from process ${msg.from}`);
        metrics[msg.from] = msg.data;
    }
});
```

And a pm2 config like this:

```
{
    "apps": [
        {
            "name": "mashroom",
            "instances": 4,
            "max_restarts": 3,
            "exec_mode": "cluster",
            "script": "starter.js",
            "env": {
                "NODE_ENV": "production"
            }
        },
        {
            "name": "metrics_collector",
            "instances": 1,
            "exec_mode": "fork",
            "script": "metrics.js"
        }
    ]
}
```

Now the worker metrics will be available under http:/localhost:15050/metrics/.

A Prometheus scrape config could look like this:

```
    - job_name: 'mashroom'
      static_configs:
        - targets:
          - localhost:15050/metrics/0
          - localhost:15050/metrics/2
          - localhost:15050/metrics/3
          - localhost:15050/metrics/4
          labels:
            service: 'Mashroom'
      relabel_configs:
        - source_labels: [__address__]
          regex:  '[^/]+(/.*)'
          target_label: __metrics_path__
        - source_labels: [__address__]
          regex:  '[^/]+/[^/]+/(.*)'
          target_label: node
        - source_labels: [__address__]
          regex:  '([^/]+)/.*'
          target_label: __address__
```

Mashroom Portal

This plugin adds a Portal component which allows composing pages from Single Page Applications (SPAs).

Registered SPAs (Portal Apps) can be placed on arbitrary pages via Drag'n'Drop. Each *instance* receives a **config object** during startup and a bunch of **client services**, which for example allow access to the message bus. The config is basically an arbitrary JSON object, which can be edited via Admin Toolbar. A Portal App can also bring its own config editor App, which again is just a simple SPA.

One of the provided client services allow Portal Apps to load any *other* App (known by name) into any existing DOM node. This can be used to:

- Create **dynamic cockpits** where Apps are loaded dynamically based on some user input or search result
- Create **Composite Apps** that consist of other Apps (which again could be used within other Apps again)

The Portal supports **hybrid rendering** for both the Portal pages and SPAs. So, if an SPA supports server side rendering the initial HTML can be incorporated into the initial HTML page. Navigating to another page dynamically replaces the SPAs in the content area via client side rendering (needs to be supported by the Theme).

The Portal also supports **i18n**, **theming**, **role based security**, a client-side message bus which can be connected to a server-side broker and a registry for **Remote Apps** on a separate server or container.

Usage

Since this plugin requires a lot of other plugins the easiest way to use it is to clone this quickstart repository: mashroom-portal-quickstart

You can find a full documentation of *Mashroom Server* and this portal plugin with a setup and configuration guide here: https://www.mashroom-server.com/documentation

The plugin allows the following configuration properties:

```
{
  "plugins": {
        "Mashroom Portal WebApp": {
            "path": "/portal",
            "adminApp": "Mashroom Portal Admin App",
            "defaultTheme": "Mashroom Portal Default Theme",
            "defaultLayout": "Mashroom Portal Default Layouts 1 Column",
            "authenticationExpiration": {
              "warnBeforeExpirationSec": 60,
              "autoExtend": false,
              "onExpiration": {
                "strategy": "reload"
              }
            },
```

```
            "ignoreMissingAppsOnPages": false,
            "versionHashSalt": null,
            "resourceFetchConfig": {
                "fetchTimeoutMs": 3000,
                "httpMaxSocketsPerHost": 3,
                "httpRejectUnauthorized": true
            },
            "defaultProxyConfig": {
                "sendPermissionsHeader": false,
                "restrictToRoles": ["ROLE_X"]
            },
            "ssrConfig": {
                "ssrEnable": true,
                "renderTimoutMs": 2000,
                "cacheEnable": true,
                "cacheTTLSec": 300,
                "inlineStyles": true
            },
            "addDemoPages": true
        }
    }
}
```

- *path*: The portal base path (Default: /portal)
- *adminApp*: The admin to use (Default: Mashroom Portal Admin App)
- *defaultTheme*: The default theme if none is selected in the site or page configuration (Default: Mashroom Portal Default Theme)
- *defaultLayout*: The default layout if none is selected in the site or page configuration (Default: Mashroom Portal Default Layouts 1 Column)
- *authenticationExpiration*:
- *warnBeforeExpirationSec*: The time when the Portal should start to warn that the authentication is about to expire. A value of 0 or lower than 0 disables the warning. (Default: 60)
- *autoExtend*: Automatically extend the authentication as long as the portal page is open (Default: false)
- *onExpiration*: What to do if the session expires. Possible strategies are *stayOnPage*, *reload*, *redirect* and *displayDomElement*. (Default: reload)

- *ignoreMissingAppsOnPages*: If an App on a page can't be found just show nothing instead of an error message (Default: false)

- *versionHashSalt*: If you need unique resource version hashes per server instance provide here a string (Default: null)

- *resourceFetchConfig*: Optional config for resource fetching (App and plugin resources like js/css files)
  - *fetchTimeoutMs*: Timeout for fetching (Default: 3000)
  - *httpMaxSocketsPerHost*: Max sockets per host for fetching resources from Remote Apps (Default: 10)
  - *httpRejectUnauthorized*: Reject resources from servers with invalid certificates (Default: true)

- *defaultProxyConfig*: Optional default http proxy config for portal apps (see below the documentation of *portal-app2* plugins). The *restrictToRoles* here cannot be removed per app, but apps can define other roles that are also allowed to access a proxy.

- *ssrConfig*: Optional config for server side rendering

- *ssrEnable*: Allow server side rendering if Apps support it (Default: true)

- *renderTimoutMs*: Timeout for SSR which defines how long the page rendering can be blocked. Even if SSR takes too long the result is put into the cache and might be available for the next page rendering (Default: 2000)

- _cacheEnable": Enable cache for server-side rendered HTML (Default: true)

- *cacheTTLSec*: The timeout in seconds for cached SSR HTML (Default: 300)

- *inlineStyles*: Inline the App's CSS to avoid sudden layout shifts after loading the initial HTML (Default: true)

- *addDemoPages*: Add some demo pages if the configuration storage is empty (Default: true)

Browser support

The Portal supports only modern Browsers and requires ES6.

Services

MashroomPortalService

The exposed service is accessible through *pluginContext.services.portal.service*

Interface:

```
export interface MashroomPortalService {
    /**
     * Get all registered portal apps
     */
    getPortalApps(): Readonly<Array<MashroomPortalApp>>;

    /**
     * Get all registered theme plugins
     */
    getThemes(): Readonly<Array<MashroomPortalTheme>>;

    /**
     * Get all registered layout plugins
     */
    getLayouts(): Readonly<Array<MashroomPortalLayout>>;

    /**
     * Get all registered page enhancement plugins
     */
    getPortalPageEnhancements(): Readonly<Array<MashroomPortalPageEnhancement>>;

    /**
     * Get all registered app enhancement plugins
     */
    getPortalAppEnhancements(): Readonly<Array<MashroomPortalAppEnhancement>>;

    /**
     * Get all sites
     */
    getSites(limit?: number): Promise<Array<MashroomPortalSite>>;

    /**
     * Get the site with the given id
     */
    getSite(siteId: string): Promise<MashroomPortalSite | null | undefined>;

    /**
     * Find the site with given path
     */
    findSiteByPath(path: string): Promise<MashroomPortalSite | null | undefined>;

    /**
     * Insert new site
     */
    insertSite(site: MashroomPortalSite): Promise<void>;
```

```
    /**
     * Update site
     */
    updateSite(site: MashroomPortalSite): Promise<void>;

    /**
     * Delete site
     */
    deleteSite(req: Request, siteId: string): Promise<void>;

    /**
     * Get page with given id
     */
    getPage(pageId: string): Promise<MashroomPortalPage | null | undefined>;

    /**
     * Find the page ref within a site with given friendly URL
     */
    findPageRefByFriendlyUrl(site: MashroomPortalSite, friendlyUrl: string): Promise

    /**
     * Find the page ref within a site by the given pageId
     */
    findPageRefByPageId(site: MashroomPortalSite, pageId: string): Promise<MashroomP

    /**
     * Insert new page
     */
    insertPage(page: MashroomPortalPage): Promise<void>;

    /**
     * Update page
     */
    updatePage(page: MashroomPortalPage): Promise<void>;

    /**
     * Insert new page
     */
    deletePage(req: Request, pageId: string): Promise<void>;

    /**
     * GetPortal App instance
     */
    getPortalAppInstance(pluginName: string, instanceId: string | null | undefined):

    /**
     * Insert a new Portal App instance
     */
    insertPortalAppInstance(portalAppInstance: MashroomPortalAppInstance): Promise<v
```

```
    /**
     * Update given Portal App instance
     */
    updatePortalAppInstance(portalAppInstance: MashroomPortalAppInstance): Promise<v

    /**
     * Delete given portal Portal App instance
     */
    deletePortalAppInstance(req: Request, pluginName: string, instanceId: string | r
}
```

Plugin Types

portal-app

Deprecated since Mashroom v2, please use *portal-app2*.

portal-app2

This plugin type makes a Single Page Application (SPA) available in the Portal.

To register a new portal-app plugin add this to *package.json*:

```
{
    "mashroom": {
        "plugins": [
            {
                "name": "My Single Page App",
                "type": "portal-app2",
                "clientBootstrap": "startMyApp",
                "resources": {
                    "js": [
                        "bundle.js"
                    ]
                },
                "local": {
                    "resourcesRoot": "./dist",
                    "ssrBootstrap": "./dist/renderToString.js"
                },
                "defaultConfig": {
                    "appConfig": {
                        "myProperty": "foo"
                    }
                }
            }
        ]
    }
}
```

A full config with all optional properties would look like this:

```
{
    "mashroom": {
        "plugins": [
            {
                "name": "My Single Page App",
                "type": "portal-app2",
                "clientBootstrap": "startMyApp",
                "resources": {
                    "js": [
                        "bundle.js"
                    ],
                    "css": []
                },
                "sharedResources": {
                    "js": []
                },
                "screenshots": [
                    "screenshot1.png"
                ],
                "local": {
                    "resourcesRoot": "./dist",
                    "ssrBootstrap": "/dist/renderToString.js"
                },
                "remote": {
                    "resourcesRoot": "/public",
                    "ssrInitialHtmlPath": "/ssr"
                },
                "defaultConfig": {
                    "title": {
                        "en": "My Single Page App",
                        "de": "Meine Single Page App"
                    },
                    "category": "My Category",
                    "tags": ["my", "stuff"],
                    "description": {
                        "en": "Here the english description",
                        "de": "Hier die deutsche Beschreibung"
                    },
                    "metaInfo": {
                        "capabilities": ["foo"]
                    },
                    "defaultRestrictViewToRoles": ["Role1"],
                    "rolePermissions": {
                        "doSomethingSpecial": ["Role2", "Role3"]
                    },
                    "caching": {
                        "ssrHtml": "same-config-and-user"
                    },
```

```
                    "editor": {
                        "editorPortalApp": "Demo Config Editor",
                        "position": "in-place",
                        "appConfig": {
                        }
                    },
                    "proxies": {
                        "spaceXApi": {
                            "targetUri": "https://api.spacexdata.com/v3",
                            "sendPermissionsHeader": false,
                            "restrictToRoles": ["Role1"]
                        }
                    },
                    "appConfig": {
                        "myProperty": "foo"
                    }
                }
            }
        ]
    }
}
```

- *clientBootstrap*: The global function exposed on the client side to launch the App (see below for an example)
- *resources*: Javascript and CSS resources that must be loaded before the bootstrap method is invoked. All resource paths are relative to *resourcesRoot*.
- *sharedResources*: Optional. Same as *resources* but a shared resource with a given name is only loaded once, even if multiple Portal Apps declare it. This is useful if apps want to share vendor libraries or styles or such. Here you can find a demo how to use the *Webpack DllPlugin* together with this feature: Mashroom Demo Shared DLL
- *screenshots*: Optional some screenshots of the App. The screenshots paths are relative to *resourcesRoot*.
- *local*: Basic configuration if the App is deployed locally
- *resourcesRoot*: The root path for APP resources such as JavaScript files and images. Needs to be relative within the package
- *ssrBootstrap*: An optional local SSR bootstrap that returns an initial HTML for the App, relative within the package (see below for an example)

- *remote*: Optional configuration if the App is accessed remotely
- *resourcesRoot*: The root path for App resources such as JavaScript files and images
- *ssrInitialHtmlPath*: The optional path to a route that renders the initial HTML. The Portal will send a POST to this route with a JSON body of type *MashroomPortalAppSSRRemoteRequest* and expects a plain *text/html* response or an *application/json* response that satisfies *MashroomPortalAppSSRResult*.
- *defaultConfig*: The default config that can be overwritten in the Mashroom config file
  - *title*: Optional human-readable title of the App. Can be a string or an object with translations.
  - *category*: An optional category to group the Apps in the Admin App
  - *tags*: An optional list of tags that can also be used in the search (in the Admin App)
  - *description*: Optional App description. Can be a string or an object with translations.
  - *defaultRestrictViewToRoles*: Optional default list of roles that have the VIEW permission if not set via Admin App. Use this to prevent that an App can just be loaded via JS API (dynamically) by any user, even an anonymous one.
  - *rolePermissions*: Optional mapping between App specific permissions and roles. This corresponds to the permission object passed with the user information to the App.
  - *caching*: Optional caching configuration
    - *ssrHtml*: Optional SSR caching configuration (Default: same-config-and-user)
  - *editor*: Optional custom editor configuration that should be used for the *appConfig* by the Admin Toolbar
    - *editorPortalApp*: The name of the Portal App that should be used to edit the *appConfig* of this App. The App will receive an extra *appConfig* property *editorTarget* of type *MashroomPortalConfigEditorTarget*.
    - *position*: Optional hint where to launch the editor. Possible values: in-place, sidebar. (Default: in-place)
    - *appConfig*: The optional appConfig the editor App should be launched with (Default: {})
  - *proxies*: Defines proxies to access the App's backend (HTTP or WebSocket)
    - *targetUri*: The API target URI
    - *sendPermissionsHeader*: Optional. Add the header *X-USER-PERMISSIONS* with a comma separated list of permissions calculated from *rolePermissions* (Default: false)
    - *restrictToRoles*: Optional list of roles that are permitted to access the proxy. The difference to using ACL rules to restrict the access to an API is that not even the *Administrator* role can access the proxy if this property is set. You can use this to protect sensitive data only a small group of users is allowed to access.
  - *metaInfo*: Optional meta info that could be used to lookup for Apps with specific features or capabilities

- *appConfig*: The default configuration that will be passed to the App. Can be adapted in the Admin App.

The *clientBootstrap* is in this case a global function that starts the App within the given host element. Here for example a React app:

```
import React from 'react';
import {render, hydrate, unmountComponentAtNode} from 'react-dom';
import App from './App';

import type {MashroomPortalAppPluginBootstrapFunction} from '@mashroom/mashroom-port

const bootstrap: MashroomPortalAppPluginBootstrapFunction = (element, portalAppSetup
    const {appConfig, restProxyPaths, lang} = portalAppSetup;
    const {messageBus} = clientServices;

    // Check if the Apps has been rendered in the server-side, if this is a Hybrid A
    //const ssrHost = element.querySelector('[data-ssr-host="true"]');
    //if (ssrHost) {
    //    hydrate(<App appConfig={appConfig} messageBus={messageBus}/>, ssrHost);
    //} else {
        // CSR
        render(<App appConfig={appConfig} messageBus={messageBus}/>, element);
    //}

    return {
        willBeRemoved: () => {
            unmountComponentAtNode(portalAppHostElement);
        },
        updateAppConfig: (appConfig) => {
            // Implement if dynamic app config should be possible
        }
    };
};

global.startMyApp = bootstrap;
```

And for an Angular app:

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {MashroomPortalAppPluginBootstrapFunction} from '@mashroom/mashroom-portal/ty
import {AppModule} from './app/app.module';

const bootstrap: MashroomPortalAppPluginBootstrapFunction = (hostElement, portalAppS

    return platformBrowserDynamic([
        {provide: 'host.element', useValue: hostElement },
        {provide: 'app.setup', useValue: portalAppSetup},
```

```
        {provide: 'client.services', useValue: portalClientServices}
    ]).bootstrapModule(AppModule).then(
        (module) => {
            return {
                willBeRemoved: () => {
                    console.info('Destroying Angular module');
                    module.destroy();
                }
            };
        }
    );
};

global.startAngularDemoApp = bootstrap;
```

In case of a Hybrid App which supports Server Side Rendering (SSR) the server side bootstrap would look like this:

```
import React from 'react';
import {renderToString} from 'react-dom/server';
import App from './App';

import type {MashroomPortalAppPluginSSRBootstrapFunction} from '@mashroom/mashroom-p

const bootstrap: MashroomPortalAppPluginSSRBootstrapFunction = (portalAppSetup, req)
    const {appConfig, restProxyPaths, lang} = portalAppSetup;
    const dummyMessageBus: any = {};
    const html = renderToString(<App appConfig={appConfig} messageBus={dummyMessageB

    return html;
    // Alternatively (supports Composite Apps)
    /*
    return {
        html,
        embeddedApps: [],
    }
    */
};

export default bootstrap;
```

The *portalAppSetup* has the following structure:

```
export type MashroomPortalAppSetup = {
    readonly appId: string;
    readonly title: string | null | undefined;
    readonly proxyPaths: MashroomPortalProxyPaths;
    // Legacy, will be removed in Mashroom v3
```

```
    readonly restProxyPaths: MashroomPortalProxyPaths;
    readonly resourcesBasePath: string;
    readonly globalLaunchFunction: string;
    readonly lang: string;
    readonly user: MashroomPortalAppUser;
    readonly appConfig: MashroomPluginConfig;
}
```

- *appId*: The unique appId
- *title*: Translated title (according to current *lang*)
- *proxyPaths*: The base paths to the proxies defined in the plugin config. In the example below the base path to the *spaceXApi* would be in *portalAppSetup.restProxyPaths.spaceXApi.*
- *resourceBasePath*: Base path to access assets in *resourceRoot* such as images
- *lang*: The current user language (e.g.: en)
- *user*: User information such as username, user display name and roles. It has the following structure:

```
    export type MashroomPortalAppUser = {
        readonly guest: boolean;
        readonly username: string;
        readonly displayName: string;
        readonly email: string | null;
        readonly permissions: MashroomPortalAppUserPermissions;
        readonly [customProp: string]: any;
    }
```

- *appConfig*: The App config object. The default is defined in *defaultConfig.appConfig*, but it can be overwritten per instance (per Admin App).

The *clientServices* argument contains the client services, see below.

Client Services

The following client side services are available for all portal apps:

- *MashroomPortalMessageBus*: A simple message bus implementation for inter app communication
- *MashroomPortalStateService*: State management
- *MashroomPortalAppService*: Provides methods wo load, unload and reload apps
- *MashroomPortalUserService*: User management services (such as logout)
- *MashroomPortalSiteService*: Site services
- *MashroomPortalPageService*: Page services
- *MashroomPortalRemoteLogger*: A facility to log messages on the server
- *MashroomPortalAdminService*: Provides methods to administer sites and pages (only available for users with the *Administrator* role)

MashroomPortalMessageBus

```
export interface MashroomPortalMessageBus {
    /**
     * Subscribe to given topic.
     * Topics starting with getRemotePrefix() will be subscribed server side via Web
     * Remote topics can also contain wildcards: # for multiple levels and + or * fo
     * (e.g. remote:/foo/+/bar)
     */
    subscribe(topic: string, callback: MashroomPortalMessageBusSubscriberCallback):

    /**
     * Subscribe once to given topic. The handler will be removed after the first me
     * Remote topics are accepted.
     */
    subscribeOnce(topic: string, callback: MashroomPortalMessageBusSubscriberCallbac

    /**
     * Unsubscribe from given topic.
     * Remote topics are accepted.
     */
    unsubscribe(topic: string, callback: MashroomPortalMessageBusSubscriberCallback)

    /**
     * Publish to given topic.
     * Remote topics are accepted.
```

```
     */
    publish(topic: string, data: any): Promise<void>;

    /**
     * Get the private user topic for the given user or the currently authenticated
     * You can subscribe to "sub" topics as well, e.g. <private_topic>/foo
     */
    getRemoteUserPrivateTopic(username?: string): string | null | undefined;

    /**
     * The prefix for remote topics
     */
    getRemotePrefix(): string;

    /**
     * Register a message interceptor.
     * An interceptor can be useful for debugging or to manipulate the messages.
     * It can change the data of an event by return a different value or block messa
     * by calling cancelMessage() from the interceptor arguments.
     */
    registerMessageInterceptor(interceptor: MashroomPortalMessageBusInterceptor): vo

    /**
     * Unregister a message interceptor.
     */
    unregisterMessageInterceptor(interceptor: MashroomPortalMessageBusInterceptor):
}
```

MashroomPortalStateService

```
export interface MashroomPortalStateService {
    /**
     * Get a property from state.
     * It will be looked up in the URL (query param or encoded) and in the local and
     */
    getStateProperty(key: string): any | null | undefined;

    /**
     * Add given key value pair into the URL (encoded)
     */
    setUrlStateProperty(key: string, value: any | null | undefined): void;

    /**
     * Add given key value pair to the session storage
     */
    setSessionStateProperty(key: string, value: any): void;

    /**
     * Add given key value pair to the local storage
```

```
     */
    setLocalStoreStateProperty(key: string, value: any): void;
}
```

MashroomPortalAppService

```
export interface MashroomPortalAppService {
    /**
     * Get all existing apps
     */
    getAvailableApps(): Promise<Array<MashroomAvailablePortalApp>>;

    /**
     * Load portal app to given host element at given position (or at the end if pos
     *
     * The returned promise will always resolve! If there was a loading error the Ma
     */
    loadApp(appAreaId: string, pluginName: string, instanceId: string | null | undef

    /**
     * Load portal app into a modal overlay.
     *
     * The returned promise will always resolve! If there was a loading error the Ma
     */
    loadAppModal(pluginName: string, title?: string | null | undefined, overrideAppC

    /**
     * Reload given portal app
     *
     * The returned promise will always resolve!
     * If there was a loading error the MashroomPortalLoadedPortalApp.error property
     */
    reloadApp(id: string, overrideAppConfig?: any | null | undefined): Promise<Mashr

    /**
     * Unload given portal app
     */
    unloadApp(id: string): void;

    /**
     * Move a loaded app to another area (to another host element within the DOM)
     */
    moveApp(id: string, newAppAreaId: string, newPosition?: number): void;

    /**
     * Show the name and version for all currently loaded apps in a overlay (for deb
     */
    showAppInfos(customize?: (portalApp: MashroomPortalLoadedPortalApp, overlay: HTM
```

```
    /**
     * Hide all app info overlays
     */
    hideAppInfos(): void;

    /**
     * Add listener for load events (fired after an app has been loaded an attached
     */
    registerAppLoadedListener(listener: MashroomPortalAppLoadListener): void;

    /**
     * Remove listener for load events
     */
    unregisterAppLoadedListener(listener: MashroomPortalAppLoadListener): void;

    /**
     * Add listener for unload events (fired before an app will been detached from t
     */
    registerAppAboutToUnloadListener(listener: MashroomPortalAppLoadListener,): voic

    /**
     * Remove listener for unload events
     */
    unregisterAppAboutToUnloadListener(listener: MashroomPortalAppLoadListener,): vo

    /**
     * Load the setup for given app/plugin name on the current page
     */
    loadAppSetup(pluginName: string, instanceId: string | null | undefined): Promise

    /**
     * Get some stats about a loaded App
     */
    getAppStats(pluginName: string): MashroomPortalLoadedPortalAppStats | null;

    /**
     * Check if some loaded Portal Apps have been update (and have a different versi
     * This can be used to check if the user should refresh the current page.
     *
     * Returns the list of upgraded Apps.
     */
    checkLoadedPortalAppsUpdated(): Promise<Array<string>>;

    /**
     * Prefetch resources of given app/plugin. This is useful if you know which apps
     * in the future and want to minimize the loading time.
     */
    prefetchResources(pluginName: string): Promise<void>;
```

```
    readonly loadedPortalApps: Array<MashroomPortalLoadedPortalApp>;
}
```

MashroomPortalUserService

```
export interface MashroomPortalUserService {
    /**
     * Get the authentication expiration time in unix time ms.
     * Returns null if the check fails and "0" if the check returns 403.
     */
    getAuthenticationExpiration(): Promise<number | null>;

    /**
     * Get the unix ms left until authentication expiration.
     * Returns null if the check fails and "0" if the check returns 403.
     */
    getTimeToAuthenticationExpiration(): Promise<number | null>;

    /**
     * Extend the authentication.
     * Can be used to update the authentication when no server interaction has occur
     */
    extendAuthentication(): void;

    /**
     * Logout the current user
     */
    logout(): Promise<void>;

    /**
     * Get the current user's language
     */
    getUserLanguage(): string;

    /**
     * Set the new user language
     */
    setUserLanguage(lang: string): Promise<void>;

    /**
     * Get all available languages (e.g. en, de)
     */
    getAvailableLanguages(): Promise<Array<string>>;

    /**
     * Get the configured default language
     */
```

```
    getDefaultLanguage(): Promise<string>;
}
```

MashroomPortalSiteService

```
export interface MashroomPortalSiteService {
    /**
     * Get the base url for the current site
     */
        getCurrentSiteUrl(): string;

    /**
     * Get a list with all sites
     */
        getSites(): Promise<Array<MashroomPortalSiteLinkLocalized>>;

    /**
     * Get the page tree for given site
     */
        getPageTree(siteId: string): Promise<Array<MashroomPortalPageRefLocalized>>;
}
```

MashroomPortalPageService

```
export interface MashroomPortalPageService {
    /**
     * Get current pageId
     */
    getCurrentPageId(): string;
    /**
     * Get the page friendlyUrl from given URL (e.g. /portal/web/test?x=1 -> /test)
     */
    getPageFriendlyUrl(pageUrl: string): string;
    /**
     * Find the pageId for given URL (can be a page friendlyUrl or a full URL as see
     */
    getPageId(pageUrl: string): Promise<string | undefined>;
    /**
     * Get the content for given pageId.
     * It also calculates if the correct theme and all necessary page enhancements f
     * are already loaded. Otherwise fullPageLoadRequired is going to be true and nc
     */
    getPageContent(pageId: string): Promise<MashroomPortalPageContent>;
}
```

MashroomPortalRemoteLogger

```
export interface MashroomPortalRemoteLogger {
    /**
     * Send a client error to the server log
     */
    error(msg: string, error?: Error): void;

    /**
     * Send a client warning to the server log
     */
    warn(msg: string, error?: Error): void;

    /**
     * Send a client info to the server log
     */
    info(msg: string): void;
}
```

MashroomPortalAdminService

```
export interface MashroomPortalAdminService {
    /**
     * Get all existing themes
     */
    getAvailableThemes(): Promise<Array<MashroomAvailablePortalTheme>>;

    /**
     * Get all existing layouts
     */
    getAvailableLayouts(): Promise<Array<MashroomAvailablePortalLayout>>;

    /**
     * Get all currently existing roles
     */
    getExistingRoles(): Promise<Array<RoleDefinition>>;

    /**
     * Get all app instances on current page
     */
    getAppInstances(): Promise<Array<MashroomPagePortalAppInstance>>;

    /**
     * Add an app to the current page.
     */
    addAppInstance(pluginName: string, areaId: string, position?: number, appConfig?

    /**
     * Update given app instance config or position
     */
```

```
    updateAppInstance(pluginName: string, instanceId: string, areaId: string | null

    /**
     * Remove given app instance from page
     */
    removeAppInstance(pluginName: string, instanceId: string): Promise<void>;

    /**
     * Get roles that are permitted to view the app (no roles means everyone is perm
     */
    getAppInstancePermittedRoles(pluginName: string, instanceId: string): Promise<st

    /**
     * Update roles that are permitted to view the app (undefined or null means ever
     */
    updateAppInstancePermittedRoles(pluginName: string, instanceId: string, roles: s

    /**
     * Get current pageId
     */
    getCurrentPageId(): string;

    /**
     * Get page data
     */
    getPage(pageId: string): Promise<MashroomPortalPage>;

    /**
     * Add new page
     */
    addPage(page: MashroomPortalPage): Promise<MashroomPortalPage>;

    /**
     * Update an existing page
     */
    updatePage(page: MashroomPortalPage): Promise<void>;

    /**
     * Delete the given page
     */
    deletePage(pageId: string): Promise<void>;

    /**
     * Get roles that are permitted to view the page (no roles means everyone is per
     */
    getPagePermittedRoles(pageId: string): Promise<string[] | null | undefined>;

    /**
     * Update roles that are permitted to view the page (undefined or null means eve
     */
```

```
    updatePagePermittedRoles(pageId: string, roles: string[] | null | undefined): Pr

    /**
     * Get current siteId
     */
    getCurrentSiteId(): string;

    /**
     * Get site with given id
     */
    getSite(siteId: string): Promise<MashroomPortalSite>;

    /**
     * Add new site
     */
    addSite(site: MashroomPortalSite): Promise<MashroomPortalSite>;

    /**
     * Update existing site
     */
    updateSite(site: MashroomPortalSite): Promise<void>;

    /**
     * Delete the given site
     */
    deleteSite(siteId: string): Promise<void>;

    /**
     * Get roles that are permitted to view the site (no roles means everyone is per
     */
    getSitePermittedRoles(siteId: string): Promise<string[] | null | undefined>;

    /**
     * Update roles that are permitted to view the site (undefined or null means eve
     */
    updateSitePermittedRoles(siteId: string, roles: string[] | null | undefined): Pr
}
```

portal-theme

This plugin types adds a theme to the Portal.

To register a new portal-theme plugin add this to *package.json*:

```
{
     "mashroom": {
        "plugins": [
           {
                "name": "My Theme",
```

```
                "type": "portal-theme",
                "bootstrap": "./dist/mashroom-bootstrap.js",
                "resourcesRoot": "./dist",
                "views": "./views",
                "defaultConfig": {
                    "param1": true
                }
            }
        ]
    }
}
```

- *resourcesRoot*: Folder that contains assets (can be accessed in the theme via *resourcesBasePath*)

- *views*: The folder with the views. There must exist a view **portal** which renders a portal page

Since *Mashroom Portal* uses the *Express* render mechanism all Template Engines supported by *Express* can be used to define the template. The bootstrap returns the template engine and the engine name like so:

```
import {engine} from 'express-handlebars';
import path from 'path';

import type {MashroomPortalThemePluginBootstrapFunction} from '@mashroom/mashroom-pc

const bootstrap: MashroomPortalThemePluginBootstrapFunction = async () => {
    return {
        engineName: 'handlebars',
        engineFactory: () => {
            return engine({
                partialsDir: path.resolve(__dirname, '../views/partials/'),
            });
        },
    };
};

export default bootstrap;
```

NOTE: Even if Express.js could automatically load the template engine (like for Pug) you have to provide the *engineFactory* here, otherwise plugin local modules can not be loaded. In that case define the engineFactory like this:
engineFactory: () => require('pug').__express

The theme can contain the following views:

- *portal*: The portal page (required)
- *appWrapper*: The wrapper for any Portal App (optional)
- *appError*: The error message if the loading of a Portal App fails (optional)

A typical *portal* view with *Handlebars* might look like this:

```
<!doctype html>
<html>
<head>
    <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fi

    <meta name="description" content="{{model.page.description}}">
    <meta name="keywords" content="{{model.page.keywords}}">
    {{#if csrfToken}}
        <meta name="csrf-token" content="{{csrfToken}}">
    {{/if}}

    <title>{{site.title}} - {{page.title}}</title>

    <link rel="stylesheet" type="text/css" href="{{resourcesBasePath}}/style.css">

    {{{portalResourcesHeader}}}

    {{#if page.extraCss}}
        <style>
            {{{page.extraCss}}}
        </style>
    {{/if}}
</head>
<body>
    <div id="mashroom-portal-admin-app-container">
        <!-- Admin app goes here -->
    </div>

    <header>
        <div class="site-name">
            <h1>{{site.title}}</h1>
        </div>
    </header>

    <main>
        {{> navigation}}
```

```
        <div id="portal-page-content" class="mashroom-portal-apps-container containe
            {{{pageContent}}}
        </div>
    </main>

    <div id="mashroom-portal-modal-overlay">
        <div class="mashroom-portal-modal-overlay-wrapper">
            <div class="mashroom-portal-modal-overlay-header">
                <div id="mashroom-portal-modal-overlay-title">Title</div>
                <div id="mashroom-portal-modal-overlay-close" class="close-button"><
            </div>
            <div class="mashroom-portal-modal-overlay-content">
                <div id="mashroom-portal-modal-overlay-app">
                    <!-- Modal apps go here -->
                </div>
            </div>
        </div>
    </div>

    <div id="mashroom-portal-auth-expires-warning">
        <div class="mashroom-portal-auth-expires-warning-message">
            {{{__ messages "authenticationExpiresWarning"}}}
        </div>
    </div>

    {{{portalResourcesFooter}}}
</body>
</html>
```

The *pageContent* variable contains the actual content with the Portal layout (see below) and the Apps.

Here all available variables:

```
export type MashroomPortalPageRenderModel = {
    readonly portalName: string;
    readonly siteBasePath: string;
    readonly apiBasePath: string;
    readonly resourcesBasePath: string | null | undefined;
    readonly site: MashroomPortalSiteLocalized;
    readonly page: MashroomPortalPage & MashroomPortalPageRefLocalized;
    readonly portalResourcesHeader: string;
    readonly portalResourcesFooter: string;
    readonly pageContent: string;
    // @Deprecated, use pageContent; will be removed in 3.0
    readonly portalLayout: string;
    readonly lang: string;
    readonly availableLanguages: Readonly<Array<string>>;
    readonly messages: (key: string) => string;
```

```
    readonly user: MashroomPortalUser;
    readonly csrfToken: string | null | undefined;
    readonly userAgent: UserAgent;
    readonly lastThemeReloadTs: number;
    readonly themeVersionHash: string;
}
```

portal-layouts

This plugin type adds portal layouts to the portal. A layout defines a areas where portal-apps can be placed.

To register a new portal-layouts plugin add this to *package.json*:

```
{
     "mashroom": {
        "plugins": [
           {
                "name": "My Layouts",
                "type": "portal-layouts",
                "layouts": {
                    "1 Column": "./layouts/1column.html",
                    "2 Columns": "./layouts/2columns.html",
                    "2 Columns 70/30": "./layouts/2columns_70_30.html",
                    "2 Columns with 1 Column Header": "./layouts/2columnsWith1columr
                }
            }
        ]
     }
}
```

- *layouts*: A map with the layout html files (on the local file system)

A layout looks like this:

```
<div class="row">
    <div class="col-md-8 mashroom-portal-app-area" id="app-area1">
        <!-- Portal apps go here -->
    </div>
    <div class="col-md-4 mashroom-portal-app-area" id="app-area2">
        <!-- Portal apps go here -->
    </div>
</div>
```

Important is the class **mashroom-portal-app-area** and a unique id element.

remote-portal-app-registry

This plugin type adds a registry for remote portal-apps to the Portal.

To register a new remote-portal-app-registry plugin add this to *package.json*:

```
{
     "mashroom": {
        "plugins": [
           {
                "name": "Mashroom Portal Remote App Registry",
                "type": "remote-portal-app-registry",
                "bootstrap": "./dist/registry/mashroom-bootstrap-remote-portal-app-r
                "defaultConfig": {
                    "priority": 100
                }
            }
        ]
     }
}
```

- *defaultConfig.priority*: Priority of this registry if a portal-app with the same name is registered multiple times (Default: 1)

And the bootstrap must return an implementation of *RemotePortalAppRegistry*:

```
import {MyRegistry} from './MyRegistry';

import type {MashroomRemotePortalAppRegistryBootstrapFunction} from '@mashroom/mashr

const bootstrap: MashroomRemotePortalAppRegistryBootstrapFunction = async (pluginNam
    return new MyRegistry();
};

export default bootstrap;
```

The plugin must implement the following interface:

```
export interface MashroomRemotePortalAppRegistry {
    readonly portalApps: Readonly<Array<MashroomPortalApp>>;
}
```

h3.

portal-page-enhancement

This plugin type allows it to add extra resources (JavaScript and CSS) to a Portal page based on some (optional) rules. This can be used to add polyfills or some analytics stuff without the need to change a theme.

To register a new portal-page-enhancement plugin add this to *package.json*:

```
{
     "mashroom": {
        "plugins": [
           {
                "name": "My Portal Page Enhancement",
                "type": "portal-page-enhancement",
                "bootstrap": "./dist/mashroom-bootstrap.js",
                "pageResources": {
                    "js": [{
                        "path": "my-extra-scripts.js",
                        "rule": "includeExtraScript",
                        "location": "header",
                        "inline": false
                    }, {
                        "dynamicResource": "myScript",
                        "location": "header"
                    }],
                    "css": []
                },
                "defaultConfig": {
                    "order": 100,
                    "resourcesRoot": "./dist/public"
                }
            }
        ]
     }
}
```

- *bootstrap*: Path to the script that contains the bootstrap for the plugin (optional)
- *pageResources*: A list of JavaScript and CSS resourced that should be added to all portal pages. They can be static or dynamically generated. And they can be added to the header or footer (location) and also be inlined. The (optional) rule property refers to a rule in the instantiated plugin (bootstrap), see below.
- *defaultConfig.order*: The weight of the resources- the higher it is the **later** they will be added to the page (Default: 1000)
- *defaultConfig.resourcesRoot*: The root for all resources (can be a local path or an HTTP url)

The bootstrap returns a map of rules and could look like this:

```
import type {MashroomPortalPageEnhancementPluginBootstrapFunction} from '@mashroom/m

const bootstrap: MashroomPortalPageEnhancementPluginBootstrapFunction = () => {
    return {
        dynamicResources: {
            myScript: () => `console.info('My Script loaded');`,
        },
        rules: {
            // Example rule: Show only for IE
            includeExtraScript: (sitePath, pageFriendlyUrl, lang, userAgent) => user
        }
    }
};

export default bootstrap;
```

The JavaScript or CSS resource can also be generated dynamically by the plugin. In that case it will always be inlined. To use this state a *dynamicResource* name instead of a *path* and include the function that actually generates the content to the object returned by the bootstrap:

```
{
     "mashroom": {
        "plugins": [
           {
                "name": "My Portal Page Enhancement",
                "type": "portal-page-enhancement",
                "bootstrap": "./dist/mashroom-bootstrap.js",
                "pageResources": {
                    "js": [{
                        "dynamicResource": "extraScript",
                        "location": "header"
                    }],
                    "css": []
                }
            }
        ]
     }
}
```

```
import type {MashroomPortalPageEnhancementPluginBootstrapFunction} from '@mashroom/m

const bootstrap: MashroomPortalPageEnhancementPluginBootstrapFunction = () => {
    return {
        dynamicResources: {
            extraScript: (sitePath, pageFriendlyUrl, lang, userAgent) => `console.ir
```

```
        }
    }
};

export default bootstrap;
```

portal-app-enhancement

This plugin type allows it to update or rewrite the *portalAppSetup* that is passed to Portal Apps at startup. This can be used to add extra config or user properties from a context. Additionally, this plugin allows it to pass extra *clientServices* to Portal Apps or replace one of the default ones.

To register a new portal-app-enhancement plugin add this to *package.json*:

```
{
     "mashroom": {
        "plugins": [
           {
              "name": "My Portal App Enhancement",
              "type": "portal-app-enhancement",
              "bootstrap": "./dist/mashroom-bootstrap.js",
              "portalCustomClientServices": {
                  "customService": "MY_CUSTOM_SERVICE"
              }
           }
        ]
     }
}
```

- *bootstrap*: Path to the script that contains the bootstrap for the plugin (could be omitted, if portalCustomClientServices is used)
- *portalCustomClientServices*: A map of client services that should be injected in the *clientServices* object the Portal Apps receive. The value (in this example MY*CUSTOM*SERVICE) needs to be an existing global variable on the page (in *window*).

The bootstrap returns the actual enhancer plugin:

```
import MyPortalAppEnhancementPlugin from './MyPortalAppEnhancementPlugin';
import type {MashroomPortalAppEnhancementPluginBootstrapFunction} from '@mashroom/ma

const bootstrap: MashroomPortalAppEnhancementPluginBootstrapFunction = () => {
    return new MyPortalAppEnhancementPlugin();
};
```

```
export default bootstrap;
```

The plugin has to implement the following interface:

```
export interface MashroomPortalAppEnhancementPlugin {
    /**
     * Enhance the portalAppSetup object passed as the first argument (if necessary)
     */
    enhancePortalAppSetup: (portalAppSetup: MashroomPortalAppSetup, portalApp: Mashr
}
```

Mashroom Portal Default Layouts

This plugin adds some default layouts to the *Mashroom Portal*.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-default-layouts** as *dependency*.

Mashroom Portal App User ExtraData

This plugin copies the property *extraData" from the server user object to the Portal App user. This is useful if the security provider adds some extra information (such as the phone number) and you want to use it in a Portal App.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-app-user-extradata** as *dependency*.

Mashroom Portal Default Theme

This plugins adds the default theme for the *Mashroom Portal*.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-default-theme** as *dependency*.

You can override the default config in your Mashroom config file like this:

```
{
    "plugins": {
        "Mashroom Portal Default Theme": {
            "spaMode": true,
            "showPortalAppHeaders": true,
            "showEnvAndVersions": true
        }
```

```
    }
}
```

- *spaMode*: The theme will try to operate like an SPA and loads new page content via AJAX and replaces the DOM. This only works until the user does not navigate on a page with a different theme or different page enhancements, in that case a full page load is triggered (Default: true)
- *showPortalAppHeaders*: Show or hide Portal App headers (Default: true)
- *showEnvAndVersions*: Show the environment (*NODE*ENV_) and version information in the header (Default: false)

Mashroom Portal Admin App

This plugin contains the default Admin Toolbar for the *Mashroom Portal*.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-admin-app** as *dependency*.

To enable it add the following to the *Mashroom Portal* config:

```
{
  "plugins": {
        "Plugin: Mashroom Portal WebApp": {
            "adminApp": "Mashroom Portal Admin App"
        }
    }
}
```

Mashroom Portal Tabify App

This *Mashroom Portal* App turns any app area where it is placed automatically into a tabbed container.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-tabify-app** as *dependency*.

After placing it on a page use the Portal Admin Toolbar to set the following properties:

- *addCloseButtons*: Defines if a close button will be added to remove a portal app displayed as tab

- *appNameTitleMapping*: A map to override the displayed title in the tab (Portal App Name -> Title to display)
- *fixedTabTitles*: A list of fixed tab titles per position (null means not fixed). E.g.: [null, 'the second tag']

Updates via MessageBus

It is possible to change the App/Title mapping and to show a specific Tab via *MessageBus*. This is especially useful for dynamic cockpits where you load Apps programmatically via *MashroomPortalAppService*.

Available topics:

- tabify-add-plugin-name-title-mapping
- tabify-add-app-id-title-mapping
- tabify-focus-app

tabify-add-plugin-name-title-mapping expect a message like this:

```
{
    pluginName: 'My App',
    title: 'Another title'
}
```

tabify-add-app-id-title-mapping expect a message like this:

```
{
    appId: '1234123',
    title: 'Another title'
}
```

And *tabify-focus-app* expects just an id:

```
{
    appId: '1234123'
}
```

Mashroom Portal iFrame App

Adds a (responsive) iFrame Portal App to the *Mashroom Portal*.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-iframe-app** as *dependency*.

After placing it on a page use the Portal Admin Toolbar to set the following properties:

- *url*: The URL of the page to show in the iframe
- *width*: The iframe width (Default 100%)
- *defaultHeight*: The height to use if the embedded page doesn't post the actual required height

To make the iFrame responsive, the embedded page has to post its height via message like so:

```
var lastContentHeight = null;

function sendHeight() {
    var contentHeight = document.getElementById('content-wrapper').offsetHeight;
    if (lastContentHeight !== contentHeight) {
        parent.postMessage({
            height: contentHeight + /* margin */ 20
        }, "*");
        lastContentHeight = contentHeight;
    }
}

setInterval(sendHeight, 1000);
```

Mashroom Portal Remote App Registry

This plugin adds a remote app registry to *Mashroom Portal*, which scans periodically a list of remote servers for Portal Apps. It expects the *package.json* and optionally an external plugin config file (default *mashroom.json*) to be exposed at */*. It also expects a *remote* config in the plugin definition, like this:

```
 {
    "name": "My Single Page App",
    "remote": {
        "resourcesRoot": "/public",
         "ssrInitialHtmlPath": "/ssr"
    }
 }
```

You can find an example remote app here: Mashroom Demo Remote Portal App .

This plugin also comes with an Admin UI extension (*/mashroom/admin/ext/remote-portal-apps*) and a REST API to add and remote URL's. The Admin UI allows adding a URL temporary only for the current session.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-remote-app-registry** as *dependency*.

You can override the default config in your Mashroom config file like this:

```
{
  "plugins": {
      "Mashroom Portal Remote App Background Job": {
          "cronSchedule": "0/1 * * * *",
          "socketTimeoutSec": 3,
          "registrationRefreshIntervalSec": 600,
          "unregisterAppsAfterScanErrors": -1
      },
      "Mashroom Portal Remote App Registry": {
          "remotePortalAppUrls": "./remotePortalApps.json"
      },
      "Mashroom Portal Remote App Registry Admin Webapp": {
          "showAddRemoteAppForm": true
      }
  }
}
```

- *cronSchedule*: The cron schedule for the background job that scans for new apps (Default: every minute)
- *socketTimeoutSec*: Socket timeout when trying to reach the remote app (Default: 3)
- *registrationRefreshIntervalSec*: Interval for refreshing known endpoints (Default: 600)
- *unregisterAppsAfterScanErrors*: Remove registered Apps of an endpoint if it cannot be reached for a number of scan intervals (Default: -1 which means: never remove)
- *remotePortalAppUrls*: Location of the config file with the remote URLs, relative to the server config (Default: ./remotePortalApps.json)
- *showAddRemoteAppForm*: Show the *Add a new Remote Portal App Endpoint* form in the Admin UI

The config file contains just a list of URLs:

```
{
    "$schema": "https://www.mashroom-server.com/schemas/mashroom-portal-remote-apps.
    "remotePortalApps": [
        "http://demo-remote-app.mashroom-server.com"
    ]
}
```

The **Service** can be used like this:

```
import type {MashroomPortalRemoteAppEndpointService} from '@mashroom/mashroom-portal

export default async (req: Request, res: Response) => {
    const remoteAppService: MashroomPortalRemoteAppEndpointService = req.pluginConte

    const remoteApps = await remoteAppService.findAll();

    // ...
}
```

The **REST API** can be used like this:

Available at */portal-remote-app-registry/api*. Methods:

- *GET /* : List of current URL's
- *POST /* : Add a new URL. Request body: `json { "url": "http://my-server.com/app1", "sessionOnly": false }`
- *DELETE /<url>* : Delete given URL

Services

MashroomPortalRemoteAppEndpointService

The exposed service is accessible through *pluginContext.services.remotePortalAppEndpoint.service*

Interface:

```
export interface MashroomPortalRemoteAppEndpointService {
    /**
     * Register a new Remote App URL
     */
    registerRemoteAppUrl(url: string): Promise<void>;
```

```
    /**
     * Register a Remote App URL only for the current session (useful for testing)
     */
    synchronousRegisterRemoteAppUrlInSession(
        url: string,
        request: Request,
    ): Promise<void>;

    /**
     * Unregister a Remote App
     */
    unregisterRemoteAppUrl(url: string): Promise<void>;

    /**
     * Find Remote App by URL
     */
    findRemotePortalAppByUrl(
        url: string,
    ): Promise<RemotePortalAppEndpoint | null | undefined>;

    /**
     * Return all known Remote App endpoints
     */
    findAll(): Promise<Readonly<Array<RemotePortalAppEndpoint>>>;

    /**
     * Update an existing Remote App endpoint
     */
    updateRemotePortalAppEndpoint(
        remotePortalAppEndpoint: RemotePortalAppEndpoint,
    ): Promise<void>;

    /**
     * Refresh (fetch new metadata) from given endpoint
     */
    refreshEndpointRegistration(
        remotePortalAppEndpoint: RemotePortalAppEndpoint,
    ): Promise<void>;
}
```

Mashroom Portal Remote App Registry for Kubernetes

Adds a remote app registry to *Mashroom Portal* which periodically scans Kubernetes services that expose Remote Portal Apps. It expects the *package.json* and optionally an external plugin config file (default *mashroom.json*) to be exposed at */*. It also expects a *remote* config in the plugin definition, like this:

```
{
   "name": "My Single Page App",
   "remote": {
       "resourcesRoot": "/public",
        "ssrInitialHtmlPath": "/ssr"
   }
}
```

You can find an example remote app here: Mashroom Demo Remote Portal App .

This plugin also comes with an Admin UI extension (*/mashroom/admin/ext/remote-portal-apps-k8s*) that can be used to check all registered Apps.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-remote-app-registry-k8s** as *dependency*.

You can override the default config in your Mashroom config file like this:

```
{
  "plugins": {
      "Mashroom Portal Remote App Kubernetes Background Job": {
          "cronSchedule": "0/1 * * * *",
          "k8sNamespacesLabelSelector": null,
          "k8sNamespaces": ["default"],
          "k8sServiceLabelSelector": null,
          "serviceNameFilter": "(microfrontend-|widget-)",
          "socketTimeoutSec": 3,
          "refreshIntervalSec": 600,
          "unregisterAppsAfterScanErrors": -1,
          "accessViaClusterIP": false,
          "serviceProcessingBatchSize": 20
      }
   }
}
```

- *cronSchedule*: The cron schedule for the background job that scans for new apps (Default: every minute)
- *k8sNamespacesLabelSelector*: Label selector(s) for namespaces, can be a single string or an array (e.g. environment=development,tier=frontend) (Default: null)
- *k8sNamespaces*: A distinct list of Kubernetes namespaces to scan; can be null if *k8sNamespacesLabelSelector* is set (Default: ["default"])

- *k8sServiceLabelSelector*: Label selector(s) for services, can be a single string or an array (e.g. microfrontend=true) (Default: null)
- *serviceNameFilter*: A regular expression for services that should be checked (case-insensitive). (Default: ".*")
- *socketTimeoutSec*: Socket timeout when trying to the Kubernetes service (Default: 3)
- *checkIntervalSec*: The time in seconds after that a registered services show be re-checked (Default: 600)
- *unregisterAppsAfterScanErrors*: Remove registered Apps of a service if it cannot be reached for a number of scan intervals (Default: -1 which means: never remove)
- *accessViaClusterIP*: Access services via IP address and not via <name>.<namespace> (Default: false)
- *serviceProcessingBatchSize*: Number of services that should be processed in parallel at a time (Default: 20)

The list of successful registered services will be available on **http://<host>:<port>/portal-remote-app-registry-kubernetes**

A more complex example

Select all services with label microfrontend=true and not label channel=alpha in all namespaces with label environment=development and tier=frontend:

```
{
  "plugins": {
      "Mashroom Portal Remote App Kubernetes Background Job": {
          "k8sNamespacesLabelSelector": ["environment=development,tier=frontend"],
          "k8sNamespaces": null,
          "k8sServiceLabelSelector": ["microfrontend=true,channel!=alpha"]
        }
    }
}
```

Priority

In case of duplicate Portal Apps the one that appears first in the list of namespaces is taken. For a configuration like this:

```
{
  "k8sNamespacesLabelSelector": ["environment=hotfix", "environment=prod"],
  "k8sNamespaces": ["namespace2"]
}
```

the order is:

- Namespaces that match *environment=hotfix*
- Namespaces that match *environment=prod*
- Namespace *namespace2*

Setup Kubernetes access

In order to allow Mashroom to fetch services for given namespaces you need to attach a Kubernetes **Service Account** with the correct permissions to the deployment.

Create a role with the required permissions like this:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: list-namespaces-services-cluster-role
rules:
  - apiGroups:
      - ""
    resources:
      - services
      - namespaces
    verbs:
      - get
      - list
```

And then create the Service Account and bind the role (we use a *ClusterRoleBinding* here so the account can read services in **all** namespaces in the cluster, if you don't want that, you have to create a *RoleBinding* per allowed namespace):

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: mashroom-portal
  namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
```

```
metadata:
  name: mashroom-portal-role-binding
subjects:
  - kind: ServiceAccount
    name: mashroom-portal
    namespace: default
roleRef:
  kind: ClusterRole
  name: list-namespaces-services-cluster-role
  apiGroup: rbac.authorization.k8s.io
```

And in your deployment resource just state the Service Account name:

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: mashroom-portal
    namespace: default
spec:
    # ...
    template:
        # ...
        spec:
            containers:
                - name: mashroom-portal
                # ...
            serviceAccountName: mashroom-portal
```

Mashroom Portal Sandbox App

This *Mashroom Portal* App can be used to load and **test any Portal App** with a specific configuration and to interact with the App via Message Bus. It can also be used for end-2-end testing with tools such as Selenium.

Usage

- If *node_modules/@mashroom* is configured as plugin path just add @**mashroom/mashroom-portal-sandbox-app** as *dependency*.
- Add the *Mashroom Sandbox App* app on an empty page
- Select the app you want to run within the sandbox

The app supports the following query parameters:

- *sbAutoTest*: This is an automated tests ann all JSON inputs should be replaced by simple textareas
- *sbPreselectAppName*: The name of the app that should be preselected (without starting it)
- *sbAppName*: The name of the app that should be started in the sandbox. If this parameter is given the app will be started automatically, otherwise all other query parameters are ignored.
- *sbWidth*: The width the app should be started with. Default: 100%
- *sbLang*: The language code that should be passed to the app.
- *sbPermissions*: The base64 encoded *permissions* object that should be passed to the app. E.g.:

```
btoa(JSON.stringify({
    permissionA: true
}))
```

- *sbAppConfig*: The Base64 encoded *appConfig* object that should be passed to the app.

For an example how to use the sandbox in an end-2-end test see: https://github.com/nonblocking/mashroom-portal-quickstart/tree/master/plugin-packages/example-react-app/test-e2e/example.test.js

Mashroom Portal Remote Messaging App

This *Mashroom Portal* App can be used to test remote messaging in the *Mashroom Portal*. This App requires the *mashroom-messaging* and *mashroom-websocket* plugins to be installed.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-demo-remote-messaging** as *dependency*.

After adding the app to a page you can send a message to another user (or another browser tab) by using the (remote) topic *user/<other-username>/*. And the app will automatically subscribe the topic *user//#* to receive all user messages.

Demo Plugin Documentation

Mashroom Demo Webapp

This a simple demo Express webapp which can be developed and run standalone, but also be integrated into *Mashroom Server* on an arbitrary (configurable) path.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-demo-webapp** as *dependency*.

After that the webapp will be available at */demo/webapp*

You can change the path by overriding it in your Mashroom config file like this:

```
{
  "plugins": {
        "Mashroom Demo Webapp": {
            "path": '/my/path'
        }
    }
}
```

Mashroom Portal Demo Alternative Theme

This is an alternative demo theme for the *Mashroom Portal*.

This theme demonstrates how to create a **type safe** theme with *express-react-views* and *TypeScript*.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-demo-alternative-theme** as *dependency*.

Mashroom Portal Demo React App 2

This is another simple React based SPA which can be developed and run standalone, but can also act as a building block in the *Mashroom Portal*.

This SPA also supports server side rendering and demonstrates how a custom editor can be used for the Portal App configuration (instead of the default JSON editor).

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-demo-react-app** as *dependency*.

Then you can place it on any page via Portal Admin Toolbar.

Mashroom Portal Demo React App

This is a simple React based SPA which can be developed and run standalone, but can also act as a building block in the *Mashroom Portal*.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-demo-react-app** as *dependency*.

Then you can place it on any page via Portal Admin Toolbar.

Mashroom Portal Demo Angular App

This is a simple Angular based SPA which can be developed and run standalone, but can also act as a building block in the *Mashroom Portal*.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-demo-angular-app** as *dependency*.

Then you can place it on any page via Portal Admin Toolbar.

Implementation hints

Angular was clearly designed for monolithic web applications that use the whole browser window. So, to make it more behave like a *Microfrontend* you have to change a few things in the template generated by the Angular CLI:

- Since you can register a *NgModule* only once, use a dummy main module that loads the actual app module (see *loader.module.ts*), otherwise you could not load the App multiple times with different configurations on the same page
- Remove all polyfills, since *Microfrontends* should not install global libraries
- Get rid of *zone.js* because it installs itself globally and might infer with other Apps

Mashroom Portal Demo Vue App

This is a simple Vue based SPA which can be developed and run standalone, but can also act as a building block in the *Mashroom Portal*.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-demo-vue-app** as *dependency*.

Then you can place it on any page via Portal Admin Toolbar.

Mashroom Portal Demo Svelte App

This is a simple Svelte based SPA which can be developed and run standalone, but can also act as a building block in the *Mashroom Portal*.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-demo-svelte-app** as *dependency*.

Then you can place it on any page via Portal Admin Toolbar.

Mashroom Portal Demo SolidJS App

This is a simple SolidJS based SPA which can be developed and run standalone, but can also act as a building block in the *Mashroom Portal*.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-demo-solidjs-app** as *dependency*.

Then you can place it on any page via Portal Admin Toolbar.

Mashroom Portal Demo Rest Proxy App

This is a simple SPA that demonstrates how the *Mashroom Portal* proxy could be used to connect to a REST API that cannot be reached directly by the client.

It fetches data *rocketlaunch.live*, but connects through the Portal. So the actual endpoint will not be visible in the browser.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-demo-rest-proxy-app** as *dependency*.

Then you can place it on any page via Portal Admin Toolbar.

Mashroom Portal Demo WebSocket Proxy App

This is a simple SPA that demonstrates how the *Mashroom Portal* proxy can be used to connect to a WebSocket server that cannot be reached directly by the client.

By default, it connects to an echo server on ws://ws.ifelse.io/, but that server might go down any time. If you can't connect, you can always launch a local WebSocket server (like https://github.com/pmuellr/ws-echo) and change the *targetUri* in *package.json* accordingly.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-demo-rest-proxy-app** as *dependency*.

Then you can place it on any page via Portal Admin Toolbar.

Mashroom Portal Demo Load Dynamically App

This is a simple SPA that demonstrates how an App registered to the *Mashroom Portal* can load and unload other Portal Apps on a page with a specific config.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-demo-load-dynamically-app** as *dependency*.

Then you can place it on any page via Portal Admin Toolbar.

Mashroom Portal Demo Composite App

This is a simple SPA that uses other SPAs (which are registered to the *Mashroom Portal*) as building blocks. We call this a **Composite App*, and it could again be a building block for other Composite Apps.

The SPA itself is written in React but is uses other ones implemented with Vue.js, Angular and Svelte to build a dialog. It is capable of server-side rendering, which includes the *embedded* Apps.

Usage

If *node_modules/@mashroom* is configured as plugin path just add **@mashroom/mashroom-portal-demo-composite-app** as *dependency*.

Then you can place it on any page via Portal Admin Toolbar.

3rd Party Plugins

- Mashroom GraphQL Plugin Loader
- Mashroom Portal Zipkin Page Enhancement