

Table of Contents

- [Table of Contents](#)
- [A Secure Web Application Template \(SWA\)](#)
 - [Why is Secure Web App useful?](#)
 - [How to get started](#)
 - [What you need](#)
 - [Installation](#)
 - [Launch](#)
 - [How it all works](#)
 - [What are modules?](#)
 - [A \(very\) simple module](#)
 - [Module Resources \(routes, views and public\)](#)
 - [Routes](#)
 - [Views](#)
 - [Public](#)
 - [How this all applies to modules](#)
 - [How to use paths with module resources](#)
 - [How to create a route](#)
 - [The module function](#)
 - [Parameters](#)
 - [Return value \(exporting\)](#)
 - [How modules are implemented](#)
 - [Default services offered by the system](#)
 - [DataAccess](#)
 - [ModuleLoader](#)
 - [Logging](#)
 - [Server](#)
 - [The module cache](#)
 - [modules_enabled.json & module load order](#)
 - [Logging in modules](#)
 - [What is meta.json](#)
 - [Included](#)
 - [Deployment](#)
 - [Tech Stack](#)
- [Maintenance & Contribution](#)
- [Code of Conduct](#)

A Secure Web Application Template (SWA)

Secure Web App template is an open-source framework for secure web applications which offers fundamental security mechanisms like secure web connection, message encryption, data integrity, and protection against various malicious attacks alongside user authentication features (register, login) out of the box, so you can get started with your next project without worrying too much about security aspects. 🛡️

It has been developed by Computer Science & Engineering students within the context of a study course at [Cologne University of Applied Sciences](#).

Why is Secure Web App useful?

It has been developed in such a way that anyone with entry-level knowledge to expert-level can benefit from it.

The power of Secure Web App lies in its modular architecture. This allows smooth addition and removal of desired functional components based on the use case and requirements.

DISCLAIMER: It is highly recommended to go over the architectural details below before typing any lines of code.

How to get started

A good starting point is to get SWA running before taking a look at how everything works. For this purpose a small chat client is included as an example on how this system could be used. Follow the steps below to install and launch SWA.

What you need

- nodejs & npm (**latest**)
- docker & docker compose (**latest**)

Note: Please make sure you got the latest versions of nodejs, npm, docker and docker compose. Otherwise the build/launch might fail.

Please refer to [nodejs](#) and [docker](#) about how to properly install this software on your operating system.

Installation

Note: It is important to follow these steps **in order**.

1. Run `npm install`
2. Followed by `docker compose build`

After these two steps the application should be ready for launch.

Launch

1. Run `docker compose up` to start all three docker containers used by the application
2. Run `npm start` to start the actual application

To verify that the application is running open a browser and navigate to localhost.

Note: If you are having connection issues you might need to open ports 80, 443 and 3307 in your firewall manually to be able to connect to the application. Note that 3307 allows for direct access to the application, bypassing the apache2 proxy. Therefore access to port 3307 should be restricted during production. This can either be done locally by defining a firewall rule that allows the apache2 proxy to connect while blocking external connections or, if you have access to it, by blocking port 3307 on the next higher level firewall (like a router firewall).

How it all works

Now that we got the application running it's time to understand how it all works. Let us start with the most important concept which you will hear a lot during this explanation: **Modules**

What are modules?

A module is a small, and for the most part independent, component of the application. Modules can vary a lot, from [simple middleware providers](#) to an entire [chat system](#).

So that's nice, but how does it work? Well it's actually very simple. It all boils down to just a couple of resources and a single function. But let's not get ahead of ourselves.

A (very) simple module

Every folder located in the [modules](#) directory is a module.

So in order to create a new module just create a new folder in [modules](#) with a descriptive name.

Great, we now have a module... except that this module is not very useful in it's current state. To actually make it useful we need to fill it with content, but how? To answer that question we need to go over the two major parts that modules are composed of: *Resources* and the *module function*. Let's start with resources.

Module Resources (routes, views and public)

If you worked with express or similar frameworks before you likely already know what **routes**, **views** and **public** refers to and can safely [skip the next three sub-chapters](#). If not, no need to worry as we will quickly explain what each of these actually mean:

Routes

To understand **routes** let's first talk about **paths**. A **path** refers to a location where something is located. If you open a web page and look at the url you will see something like <https://somepage.com/example/thatOneFile.html>. The part that is highlighted in **bold**? That is called a **path** and refers to a resource: A file called [thatOneFile.html](#).

Now this is useful and all, but it's also a bit limiting in what you can do. Think about it, this **.html** file is 'finished', there is nothing we can really do with it. But what if we want to make a web application, something a lot more complex than a simple page?

How about this? Once our server receives a request for [example/thatOneRoute](#), instead of just stupidly responding with some file located at that **path**, we execute some code just for this one **route**, this code can

then send whatever it wants to the client. This is the basic concept behind **routes**.

So now we got a request to our newly created **route** and we want to send a response page, but writing html in strings is a bit cumbersome, isn't there a better solution? Yes, that's where **views** come into play.

Views

A **view** is really just an html file, or a file that can be turned into html by some sort of rendering engine. This view can then conveniently be sent by our code located at our **route** without needing to define long html in strings.

Now we could just send this html file and be done with it, but can we do more? One thing we lost by introducing **views** is the ability to customize our html freely to fit our needs just for this specific situation. To remedy this issue and get the best of both worlds we introduce another concept: **Template engines**.

A template engine allows us to process our **view** before sending it to the client. We can, for instance, use this to insert an anti-csrf token into our html.

Our system uses **ejs** by default, but other engines can be applied if needed.

We now know what **routes** and **views** are, but isn't it all a bit much. Creating a **route** and defining a **view** quickly gets very mundane if we just want to send an image for instance. Sometimes a stupid 'send what can be found at this **path**' server is just better. That's why we got a third type of resource: **public**

Public

public files have a very simple concept: If requested just send this file to the client. No fancy processing, no customization based on the current situation, just send that file.

You should consider however that this also applies to protections you might want your files to have. Everything marked as **public** is **public** and can be read by everyone.

How this all applies to modules

So now we learned what **routes**, **views** and **public** files are (or just skipped to this point). But how does this knowledge help us develop our module?

It's simple. A module can offer all these three things as resources without us having to write any net-code. And all we need to do is to create three more folders in our module called:

- **routes**
- **views**
- **public**

Now we simply move each file in the corresponding folder and voila, were done... well almost. If you've been following along you know that one of these categories is not just a simple file, but actual code; and naturally we need to talk about how to actually implement said code.

But first, let us cover how **paths** work within this structure.

How to use paths with module resources

Say we place an image `user.png` in public (`public/user.png`), then we can access it with `https://localhost/user.png`.

But what if we want it to be accessible under `https://localhost/img/users/user.png` instead? Well, just rebuild the path locally using folders: `public/img/users/user.png`. Now we can access our image exactly where we want it to be available. However this works not only for images or even just `public` files, but for all the three types of resources we have. Suppose we have the route `chat.js`, then we can place the route in say `routes/chatting/chat.js` to make it accessible under `https://localhost/chatting/chat` (note that the `.js` suffix is removed for routes).

How to create a route

A route in your module is represented by a single javascript file placed inside the `routes` folder.

Every `route` (meaning every javascript file placed in `routes`) has to export a `router`. This `router` can then react to all incoming requests targeted at this specific `route`.

As an example look at this route file: `logout.js`. In the first two lines a new `router` is constructed. The next section defines to what requests this router should react to. In this example it only reacts to `GET` requests sent to the `routes` index (`/`) page. Lastly the router is exported.

Every `route` should generally follow this structure. First create a new router, then set-up the router to react to certain methods & targets, then export the router.

Now let's go back to the example above, because there is still something important to discuss: Which `path` will lead us to the router? One might think it's `https://localhost/`, since we listen on index (`/`). But remember what we learned in the [previous chapter](#) about how pathing works in a resource folder. We learned that resources are accessible on the server at the location they are placed in the resource folder.

Since our folder structure is `routes/logout.js` the `route` becomes accessible at `https://localhost/logout` (remember that `.js` is removed from routes). But what happens with the `path` defined in `router.get()`? Simple, we join them together. This means that we have `https://localhost/logout + /`, resulting in `https://localhost/logout/` or just `https://localhost/logout` since a trailing `/` has no meaning in url paths.

There is one special exception to this rule though: If the route file is called `index.js` it is resolved to `/`. So `routes/api/index.js` would become `https://localhost/api/` on the server and `routes/index.js` would become `https://localhost/`, thereby becoming the server's index page.

To summarize all this information:

- A `route` has to export a `router`
- This `router` can have an multiple `routes` it reacts to.
- The `server` path is just the `local` path + the `router` path
- An `index.js` route file resolves to `/`

The module function

Now that we learned about module resources let us take a look at the second major part of a module: The module function

What even is a module function? A module function can be understood as the entry point into a module. Imagine C's `main()` function, this is a similar concept. Another way to describe a module function is calling it an 'on load' listener or callback, as this function is called once the module is loaded.

But this function is a lot more than just a simple listener. To understand why we have to look at what parameters it receives and how its return value is handled.

Parameters

A module function accepts four parameters, these are in order:

1. `app`: The `express application` object
2. `cache`: The `module cache`
3. `logger`: The module specific `logger` object
4. `module`: The own module object (see `module cache`)

Among these the most important one is the `express application`, as this is the earliest instance this object can be obtained within the module. This allows to for example install global `middleware` functions or to configure the express application. This would be impossible for a module without obtaining the application object.

The other three parameters are useful, but can also be obtained through different ways like this:

- `cache === require(process.modules)`
- `logger === process.logger(<module-name>)`
- `module === cache.get(<module-name>)`

Just this alone makes the module function quite important. The ability to modify and extend the express application object massively increases the things a module can do. But this is still not all the module function does, because there is still the return value.

Return value (exporting)

A module function can return pretty much everything. The interesting part is what happens with these returned values, because the module loader does not just discard these. Instead they are stored inside a special attribute of the module object called `exports`. This exports object can then be retrieved by other modules using the module `cache`. This allows to export values from one module to another module.

This can also be used to create overwritable stubs and interfaces. An example of this concept can be found in the `hashing` module. This module does not actually implement hashing functions by itself, instead it exports two stubs which are then overwritten by the `hashing.bcrypt` module with `bcrypts` hashing functions.

A module function can also return a `Promise` which can later be resolved. This allows for module functions to be marked as `async`. If a `Promise` is returned module loader will wait until it is resolved before beginning to load the next module.

How modules are implemented

Now that we know what modules are it is time to learn how to properly implement a module. Let us first discuss what your core system offers to any module, in other words what our basic toolset is. We already

know that express is part of it, but in addition to express the application offers a couple of services to aid in development.

Default services offered by the system

The following services can be accessed by all modules at all times:

DataAccess

The **DataAccess** service allows for interactions with an underlying mysql database. The connection specification used to connect to the mysql database server can be found in `.env` under `app database`. Note that these values should not be changed unless the deployment structure is subject to change as well (for a more detailed breakdown of the deployment structure of the application see [Deployment](#)).

DataAccess offers an interface called `db.js` located in `src/database`, which can be imported using `require(process.database)`.

ModuleLoader

The **ModuleLoader** service is responsible for loading and managing all modules. More module development especially important is the module cache, a global cache keeping track of all loaded modules allowing for dependencies between modules.

The global cache can be obtained using the `modules.js` interface, which is located in `src/modules` and can be imported using `require(process.modules)`. It is also passed to the module function as the `cache` parameter. The module cache is further discussed in the [next chapter](#).

Logging

The **Logging** service offers quick and easy to use loggers which should always be preferred over printing messages directly to `stdout`. **Logging** allows for namespaces, different log levels, log level filtering and colorization of output based on log level.

When a module is first created, a logger for said module is created too with it's namespace equal to the module name. This logger is then passed to the module function as the `logger` parameter. This logger is also referenced in the module object as the `logger` attribute. Lastly, a logger can also be obtained anywhere in the application by using `process.logger(<namespace>)`. `process.logger()` stores all created loggers in a cache. This means that by calling `process.logger()` with the same namespace the exact same logger object is returned.

Server

The **Server** component creates and manages the actual server the application is running on. It also creates the express application. Since modules do not interact with the server itself this service does not offer any importable interfaces. In case the server needs to be modified or extended: All code related to the server itself is located in `src/server.js` and all code related to the creation of the express application can be found in `src/app.js`.

The module cache

The module cache keeps track of all loaded modules, indexed by their module name. The `ModuleCache` class inherits `Map` and therefore offers all functions offered by `Map` as well. In addition, a new function called `require` is included (not to be confused with the `require` function offered by nodejs). This function behaves differently from `Map.get()` in that it

- returns the `exports` attribute of a module rather than the module object itself
- throws an exception if a module is `required` that is not loaded (this behavior can be suppressed by passing `true` as a second parameter)

Using `cache.require()` is the preferred way to access exports of other modules.

Module developers are encouraged to provide information about necessary dependencies of their modules in the module's `meta.json` file.

modules_enabled.json & module load order

An important part of the module system is `modules_enabled.json`. This file tells the module loader two things: What modules should be loaded and in which order modules should be loaded in.

A module is not loaded unless its module name (determined by the name of its root directory) is placed somewhere into the `include` array of `modules_enabled.json`.

The order in which modules appear in the `include` array determines the load order of the modules. The following rules apply:

1. If module `x` requires exports of another module `y`, then `x` needs to be loaded after `y` (appear later in `include`)
2. If resource `r` of module `x` should have priority over resource `r` of module `y` (that is, should overwrite resource `r` of `y`), then `x` needs to be loaded after `y` (appear later in `include`)

As an example let's look at the `hashing` module from earlier. If you look at the module order in `modules_enabled.json`, you should notice that `hashing.bcrypt` appears after `hashing` in the `include` array. This is because `hashing.bcrypt` needs to import the stubs that `hashing` exports, so the first rule applies.

Another example is the `error404` module. All this module does is to react to any request and send an error page. It should work like this: If no other module reacted to this `route` before, then 404 not found should be responded. Now where do we need to place this module in `include`? Since we are concerned with module resources the second rule applies, where `error404` is `y` and any other module with resources is `x`. This means that we need to place `error404` before any other module with resources in `include`, so that other resources can take priority over `error404`.

Logging in modules

As explained [here](#), our system offers a logging service. All modules should use this service to send log messages to `stdout`.

To do this all modules receive a `logger` object. This logger is preconfigured to use the module name as namespace, allowing for easy identification of the origin of a log event. These loggers are stored globally and can be retrieved with `process.logger(<module-name>)`.

Loggers are very simple objects. All they offer are five functions, each sending a log event at a different log level and taking a string as their only parameter. These are from most to least significant:

- **error**: These events should be used exclusively for serious problems (like thrown errors or unexpected situations that cannot be handled by the system). If an **error** event was logged that means the task could not be completed and is either unfinished or in an error state.
- **warn**: These events can be understood as less serious **error** event. One common application of **warn** events is to notify about a sub-task having failed to execute, triggering a fail-safe at the main task.
- **info**: These events should be used for information a person not interested in debugging the application could find interesting. Note that reduction of **info** events is generally recommended and more often than not an **info** event would have better been logged as a **verbose** event.
- **verbose**: These events should log everything that could be interesting, like a failed login attempt for instance. This is the first log level where the general mentality of 'more is better' applies.
- **debug**: These events should log everything needed to debug the application. Unlike **info** events maximizing debug events is usually the correct approach.

What is meta.json

meta.json is a file that should be located in your module next to your **module.js**. This file provides meta data about your module. This data should include the author of the module, a short description, the module version and what dependencies your module needs.

While there is no enforced ruleset for the structure of **meta.json** yet, we highly recommend to respect the following layout:

- **author**: The author of the module, either a single developer or the developer's group / organization
- **version**: The module version
- **description**: A short description of the module
- **depends**: An array of module names listing modules that should be loaded before this module
- **prepends**: An array of module names listing modules that should be loaded after this module
- **requires**: An array of module names listing modules those resources this module requires during runtime. This differs from **depends** and **prepends** in that this key should not imply a specific loading order, just that these modules should be loaded at some point.
- **<custom-key>**: Any custom key-value pair that might be useful to document or to configure the module

meta.json can be accessed during runtime via the **meta** attribute of the module object. Note however that changed values for **meta** are not written back to **meta.json** and will be discarded after a restart.

Included

Below is a list of all significant modules already included in the application:

- **login**: A simple login module that can be extended or just used as is. This module also

Deployment

A note about the current deployment: SWA is indented as a template. At it's core stands the nodejs application which can, in theory, be deployed in a lot of different contexts. You may want to customize your

deployment structure to fit your personal needs.


Tech Stack

SWA uses an apache2 web server as reverse proxy to communicate with the client. This proxy is responsible for establishing a secure TLS connection to the client. Behind the proxy traffic is sent unencrypted to the SWA server. This proxy is contained within a docker container managed by docker compose

SWA itself runs in an npm-managed nodejs environment.

SWA uses two databases, one serves as the main storage area, the other is used by express to save session data. Both are represented as two different mysql server in two separate docker containers managed by docker compose.

Maintenance & Contribution

This project is conducted without considering a regular maintenance. However, we welcome feedback and PRs. 

Code of Conduct

TODO for GitHub release