



# B5 - Application Development

B-DEV-500

## Dashboard

All your information in the blink of an eye



2.0



# Dashboard

language: Java, .NET, node.js

compilation: docker-compose build && docker-compose up (cf. **Project Construction**)



- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

## GENERAL CONSIDERATIONS

As part of this project, you will take on the role of a Software Architect.

Your main goal is neither reinventing the wheel nor writing numerous lines of code. On the contrary, your main action is to understand, select and integrate a wide range of existing libraries.

The code you write will only implement the so-called *business* logic. In other words, your main job will be to write *glue* between selected software components to complete the requested project.

Before embarking on the carrying out of such a project, we suggest you take the time to analyze and understand the operation of each software brick. In the idea, we will talk about **state of the art** and **POC**:

- **State of the art**: Study the different possible solutions and choose the right component according to what is needed.
- **POC (Proof Of Concept)**: Make a quick demo program that proves the proper functioning of a component or algorithm.



Your active participation in the preparation of *Workshops* and your attention during these activities will be **essential** for the success of this project.



## THE PROJECT

---

The purpose of this project is to familiarize you with the software platform you have chosen (Java, .NET, node.js) through the creation of a dashboard.

To do this, you need to implement a web application that works like [Netvibes](#).

## FEATURES

---

The app will offer the following features:

- The user registers on the application in order to obtain an account (cf *User management*)
- The registered user then confirms their enrollment on the application before being able to use it (see *Authentication / Identification*)
- The application then asks the authenticated user to subscribe to **Services** (cf *Services*)
- Each **Service** offers **Widgets** (see [Widgets](#))
- The authenticated user composes his **Dashboard** by inserting **previously configured widget instances** (see [Dashboard & widget Instance](#))
- A **Timer** allows to refresh the information displayed by the different **widget instances** present on the **Dashboard** (see *Timer*)

## WORK GROUP

---

The project is to be carried out in a group. Validation of the associated unit will take into account not only the quality of the work accomplished but also the quantity of available features.

Here is the minimum expected configuration for a group of X students:

- Let NBS be the number of **Services** supported by your **web application**
- NBW is the total number of **Widgets** supported by all available **Services**

The following conditions must be respected:

- $NBS \geq 1 + X$
- $NBW \geq 3 * X$

## USER MANAGEMENT

---

Since the application focuses on the information users are viewing, it must offer some sort of management of this information.

To do this, you need to create a user management module.

The web application offers unauthenticated users to register via a form. Take inspiration by what you already know about this step (eg fill in an email address, registration via a third party service like Yammer, Facebook, Twitter, etc.).



An administration section would be useful to manage site users.

## AUTHENTICATION / IDENTIFICATION

---

Using the application requires knowing the user in question.

To do this, it is necessary to implement the following options:

- A method of user authentication via a username / password.
- A method of identifying users via [OAuth2](#) (eg Yammer / Twitter / Facebook / etc.)



Regarding the method of identification, remember to connect the third party account to a system user.

## SERVICES

---

As the purpose of the application is to display information from different **Services** (Intra Epitech, Outlook 365, Yammer, OneDrive, etc.), it is first necessary to ask the authenticated user to select the **Services** for which they have an account.

In this part, it will be necessary to ask users to subscribe to these **Services** (eg from their profile page, the user needs to fill in their Intra Epitech credentials, the user links their Twitter / Google account / etc via an authentication [OAuth2](#)).

The **Available Services** offered to the user corresponds to the list of **Services** managed by your web application, that is to say the list of **Services** offering **Widgets**.



Some **Services** that do not require identification, for example a **Weather Service**, will be available by default for any authenticated user, others will have to be hidden until the user connects via the app.

## WIDGETS

---

Each **Service** will offer **Widgets**. **Widget instances** can be added to the **Dashboard** after being configured.

Some examples :

- **Service Weather**
  - Temperature display for a city V
- **Service Exchange**
  - Display the exchange rate of a currency pair M1 / M2
  - Display of the evolution of the price of an A share
- **Service Cinema**
  - Display of the list of the day's screenings for cinema C
- **Service RSS**
  - Display the list of the last N items for the F feed
- **Steam Service**
  - Display of the number of players for the game J
- **Service Google Map**
  - Display of the journey between the current location and an A address using MT means of transport
- **Service Youtube**
  - Display of the number of subscribers for the chain C
  - View number of views for video V
  - Displaying the last N comments for video V
- **Service Reddit**
  - Display of the last N posts for subreddit S



A valid **Widget** is a **Widget** that offers a configuration when it is inserted on the **Dashboard**. The “The YouTube viewing trends” **Widget** does not offer a configuration and is therefore not a valid **Widget** and will therefore not be accounted for.

## DASHBOARD & WIDGET INSTANCE

---

The main page of the application represents the **Dashboard**. From the latter, the authenticated user can:

- Add a new **Widget** instance by asking the user to:
  - Select a **Widget**
  - Configure the previously selected **Widget**
  - Enter the preconfigured refresh rate of the **Widget**
  - Add the **Widget** on the **Dashboard**
- Reconfigure a **Widget instance** present on the **Dashboard**
- Move a **Widget instance** present on the **Dashboard**
- Delete a **Widget instance** present on the **Dashboard**



A **Widget** is valid if it allows to propose at a time T, two **instances** of the same widget which give different information.

## TIMER

---

This essential element of the application aims to refresh the information presented by the **Widget** instances present on the **Dashboard**.

To do this, for each **Widget** instances present on the **Dashboard** will communicate to **Timer** its refresh rate so that it triggers the necessary updates at the right time.



## PROJECT CONSTRUCTION

---

Considering the large amount of leeway being given as part of this project, you will be free to use the technologies of your choice, while respecting the following guidelines:

- for Java: use of `gradle` for dependency management and compilation
- for .NET: use `nuget` and `dotnet` for dependencies and compilation
- for NodeJS: use of `npm` for the dependencies

In order to homogenize the construction of such a project you will also have to rely on the use of `Docker Compose`



Please read the following points carefully

### DOCKER-COMPOSE BUILD

---

You will have to make a `docker-compose.yml` file at the root of your project, which will describe the different docker services used.

This file must include at least the `Docker service` **server** used to launch the application on port 8080 .

### DOCKER-COMPOSE UP

---

The validation of the integrity of your application will be done when launching the `docker-compose up` request.

The following points should be respected:

- The **server** service will run by exposing the `port 8080`
- The **server** service will respond to the request `http://localhost:8080/about.json` (see **File about.json**)





## FILE ABOUT.JSON

---

The **server** will have to answer the call `http://localhost:8080/about.json`.

```
{
  "customer": {
    "host": "10.101.53.35"
  }
  "server": {
    "current_time": 1531680780,
    "services": [{
      "name": "weather",
      "widgets": [{
        "name": "city_temperature",
        "description": "Display temperature for a city",
        "params": [{
          "name": "city",
          "type": "string"
        }]
      }]
    }, {
      "name": "rss",
      "widgets": [{
        "name": "article_list",
        "description": "Displaying the list of the last articles",
        "params": [{
          "name": "link",
          "type": "string"
        }],
        {
          "name": "number",
          "type": "integer"
        }
      }]
    }
  ]
}
```

The following properties are required:

- **client.host** indicates the IP address of the client performing the HTTP request
- **server.current\_time** indicates the server time in the [Epoch Unix Time Stamp](#) format
- **server.services** indicates the list of **Services** supported by the server
- **server.services[].name** indicates the name of the **Service**
- **server.services[].widgets** indicates the list of **Widgets** supported by this **Service**
- **server.services[].widgets[].name** indicates the identifier of this **Widget**
- **server.services[].widgets[].params** indicates the list of parameters to configure this **Widget**
- **server.services[].widgets[].params[].name** indicates the identifier of this parameter
- **server.services[].widgets[].params[].type** indicates the type of this parameter. Supported types are: integer, string



## DOCUMENTATION

---

It is important to take the time to define a simple architecture without code duplication and that is also scalable.

You are asked to provide clear and simple documentation for your project. You can integrate some design schemes: class diagram, sequence diagram ...

The goal is to have a document that serves as a working tool to easily understand the project in order to facilitate communication in the work teams and facilitate the development of skills of new developers. Thus, there is no need to make class or sequence diagrams of the entire project, but rather to choose the important parts to understand that need to be documented.



You will need to provide a README.md file describing the list of **Services** and **Widgets** available on your application. This file must be written using the [markdown format](#).

## BONUS

---

The widgets exploiting the school tools (Intra Epitech, Yammer, Office 365, OneDrive, etc.) will be better used for this project.

Regarding the deployment part, particular attention will be paid to the use of cloud services. Here is a list, not exhaustive, of sites to consider for deployment:

- [Heroku](#)
- [Microsoft Azure](#)
- [Amazon AWS](#)
- [Google Cloud Platform](#)



Have Fun!