J. R. Almeida, L. Coelho and J. L. Oliveira

BIcenter: A collaborative Web ETL solution

Contents

List of Figures

List of Tables

Preface

1

Introduction

BIcenter is a web-based platform that allows the building and management of ETL pipelines, by non-IT users, in a multi-institution environment. Each institution manages and mantains ETL tasks and provides the resources for the execution of the associated tasks. Thus, each institution owns its private data sources, servers for ETL task execution and a task scheduler that allows periodic execution. In order to provide access and management control of ETL tasks and institutions, there are four distinct types of users:

- Administrator: Entity that moderates the platform. This actor has permissions to create and delete institutions.
- Resource Manager: Entity that manages private data sources and execution servers. This actor has permissions to create and delete private data sources and execution servers, within specific institutions.
- Task Manager: Entity that builds and executes ETL tasks. This actor can create and configure ETL tasks, within specific institutions.
- Data Analyst: This actor has permissions to inspect task execution history, namely the resulting data, execution logs and performance metrics.

1.1 Main requirements

Information Security Since ETL tasks parse and handle sensitive data that belongs to a particular institution, the system must be designed and implemented taking in account these security issues, namely user authentication, access control, data protection and isolation.

System Reliability Considering the periodic execution of ETL tasks, it is important to ensure that each execution is correctly initialized, started, motorized and concluded. When some fatal error occurs during an ETL

task execution, the system must be able to handle the error and successfully conclude the execution.

Solution Scalability Since a complete ETL tool typically encompasses a wide variety of components, it is crucial to build an agile approach to the development and integration of new ETL components.

2

System Implementation

This chapter contains the description of the main aspects of BIcenter implementation.

2.1 Requirements

The main goal of this application was to allow the building and management of ETL pipelines, by non-IT users, in a multi-institution environment. Therefore, some functional requirements were were considered to implement a proper and reliable solution.

Each institution manages and mantains ETL tasks and provides the resources for the execution of the associated tasks. Thus, each institution owns its private data sources, servers for ETL task execution and a task scheduler that allows periodic execution. In order to provide access and management control of ETL tasks and institutions, there are four distinct types of users: 1) Administrator, entity that moderates the platform, and it has permissions to create and delete institutions; 2) Resource Manager, entity that manages private data sources and execution servers, and it has permissions to create and delete private data sources and execution servers, within specific institutions; 3) Task Manager, entity that builds and executes ETL tasks, and it can create and configure ETL tasks, within specific institutions; and 4) Data Analyst, actor with permissions to inspect task execution history, namely the resulting data, execution logs and performance metrics.

The functional model was derived from the identification of all key components from a top-down analysis of the system requirements. Figure 2.1 illustrates step by step all actions and interactions between the main actors in order to build and execute ETL tasks and to analyze the output results. Firstly, the administrator must create an institution. A resource

manager with access to the institution can create and configure all desired private data sources and task execution servers. Thereafter, a task manager can build and configure the ETL task and then schedule it for periodic or non-periodic execution. At the configured times the task will be sent to the execution server and executed. At the end of the execution, a notification will be sent to the data analysts, so they can check the execution results.

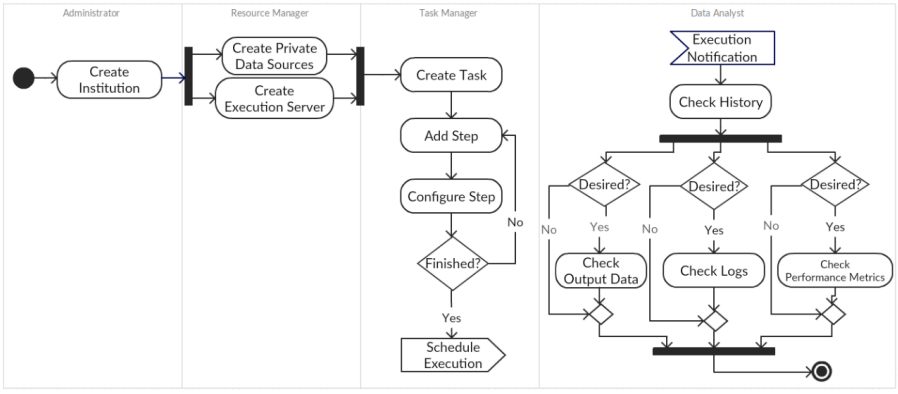

FIGURE 2.1: System workfow to build, manage and execute ETL tasks.

2.2 Architecture

The architecture considers four different tiers: 1) Application tier controls all application functionalities and maintains the system business logic; 2) Data tier is responsible for the maintenance of the private data sources; 3) Processing tier has the duty of executing and monitoring ETL task executions; and 4) Client tier is responsible for the solution presentation and page rendering. Figure 2.2 illustrates this architecture.

The system logic is built upon five different controllers: 1) RBAC controller provides user identity and evaluates access requests to resources; 2) Institution controller allows the creation and destruction of institutions, and resource allocation in institutions; 4) Task controller enables the creation, configuration and destruction of ETL tasks; and 5) Execu-

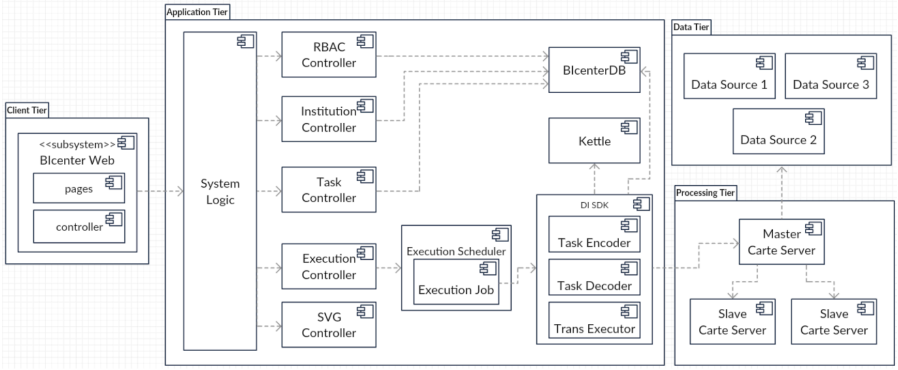

FIGURE 2.2: Components diagram of system architecture.

tion controller uses Kettle and Carte servers, so it can build, remotely execute and monitor ETL tasks.

Data Integration (DI) Software Development Kit (SDK) seeks to bridge the gap between Kettle and the stored information in BIcenter database (DB). It provides methods to build Pentaho's ETL processes according to the stored information, and also to execute them. Nevertheless, task execution is a periodic process. Therefore, Execution Scheduler manages the task execution scheduling. When the appropriated time arrives, a Execution Job is triggered. That job will communicate with DI SDK so that the given task is indeed executed. SVG controller maps between images and components, with the intent to build the visual pipeline.

BIcenter Web client adopted Model-View-Controller (MVC) architectural pattern. This pattern divides the web application in three different components:

- Model: represents the knowledge and data in an application. It has the responsability to respond to information requests, proceed to information changes according to given instructions requests, and to notify observers in event-driven systems when information changes. Typically, the application data is stored in a database.
- View: it represents the user interface. The View updates the UI upon changes in the Model, by rendering the data into the suitable UI form.
- Controller: it handles events that occurs in the View, such as user interactions, and updates the Model accordingly.

2.3 ETL SDK

The ETL Task Editor was idealized to allow users to build a visual representation of the desired ETL pipeline. This editor must be able to process a similar structure to the ones used by the desktop ETL tools and generate the correspondent visual representation of the ETL task. MxGraph was used to build the ETL pipeline editor, and this Java/JavaScript diagramming library enables the building of interactive graphs. A graph consists in a set of cells. A cell can either be a vertex or an hop. Thus, a graph is formed by a group of vertices connected by edges. The vertices correspond to the ETL steps and the edges correspond to the ETL hops. Hence, in order to exhibit the desired task within the mxEditor, it is necessary to translate the task to the corresponding mxGraph object. GraphDecoder is responsible for creating a mxGraph and defining the appropriated model, according to the given Task. To insert vertices and edges in the graph model, a transaction must be created (beginUpdate and endUpdate). This is required for the model to remain in a consistent state. A default parent is automatically created and represents the first child of the root cell in the model. Subsequent elements must be added to the default parent.

The ETL SDK was built on top of PDI SDK. Kettle contains a rich set of data integration functionality that is exposed in a set of data integration tools. However, we can also use Kettle as a library in our own software and solutions. Pentaho Data Integration can be used as a Java API composed by three main components: 1) Core, that contains the core classes for Kettle; 2) Database, that contains the database-related classes; and 3) Engine, that contains the Kettle's runtime classes.

Figure 2.3 explains step by step how an ETL task is executed. The initialization of the Kettle environment loads all available plugins, initializes the logging environment and set up and reads the system variables. After the environment initialization, the transformation metadata is loaded and a transformation engine object is instantiated. Then, the execution is prepared and the transformation threads are started. Finally, because the whole transformation runs multi- threaded, it waits until all processing is completed.

In Kettle, an ETL process can be represented by six classes [31]:

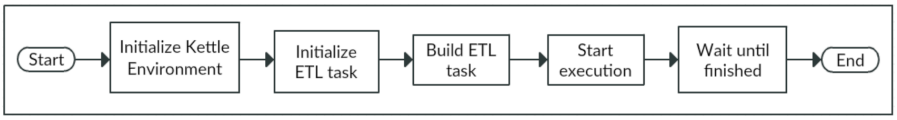

FIGURE 2.3: ETL task execution flowchart.

- TransMeta: is the class that defines the information about the ETL process and offers methods to save and load it from XML, as well as methods to alter an ETL process by adding/removing databases, steps, hops, etc.
- Trans: represents the information and operations associated with the concept of an ETL process. It can loads, instantiates, initializes, runs, and monitors the execution of the ETL process.
- DatabaseMeta: defines the database specific parameters for a certain database type.
- StepMeta: is the class that defines the information about a ETL process's Step.
- TransHopMeta: defines a link between two steps in an ETL process.
- BaseStep: represents the information and operations associated with the concept of an ETL process Step. It offers initialization, row processing and step clean-up methods.

Initially a TransMeta must be properly configured with all proper DatabaseMetas, StepMetas and TransHopMetas. TransDecoder is responsible to create a TransMeta and define all underlying information, according to the given Task. GenericStep is a class that has a generic algorithm to encode or decode a given Step to or from a Kettle's StepMeta. In order to provide interface segregation, TransDecoder must use the StepDecoder interface that only provides the GenericStep decoding method.

After having a correctly configured TransMeta, it can be executed with the associated Trans object. TransExecutor is a singleton responsible for the initialization, execution and monitoring of the Trans object. Moreover, it writes and stores the execution logs, step measures and status, and the resulting data in real-time. To accomplish this, listeners are coupled to the Trans and Steps objects in order to obtain the results and metrics throughout the transformation's execution.

3

Software User Manual

This chapter presents the main interfaces of BIcenter, aiming to help users to navigate through the system.

3.1 Initial views

The login page presented in Figure 3.1 allows users to be authenticated using their credentials. This can be connected with the institution LDAP, which provides a central place to store usernames and passwords.

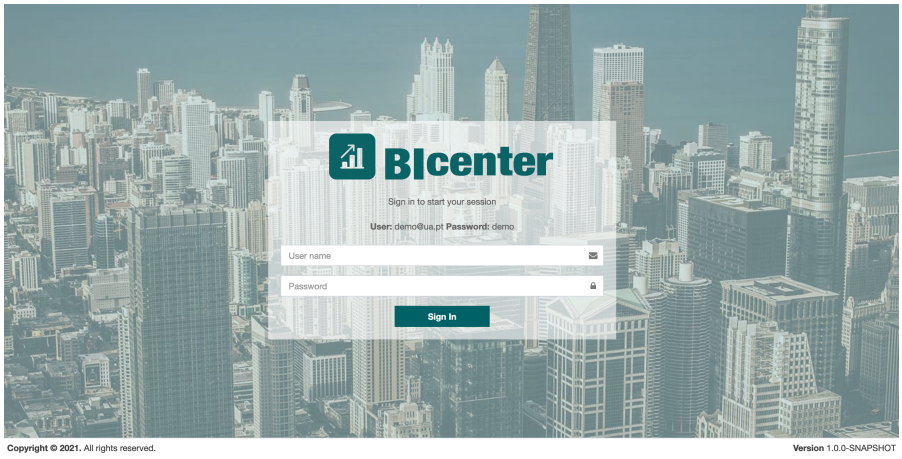

FIGURE 3.1: Login page.

After the authentication process, the user is redirected to the application's home page represented in Figure 3.2. This page contains all the institutions that the user belongs to. In this case, this user has permission to create a new instance of an institution in the application using the modal illustrated in Figure 3.3.

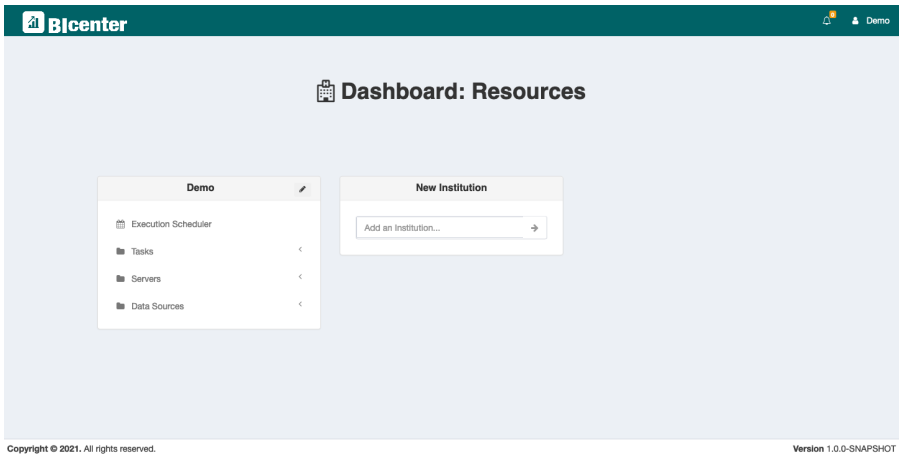

FIGURE 3.2: Home page after login with all institutions assigned to the user.

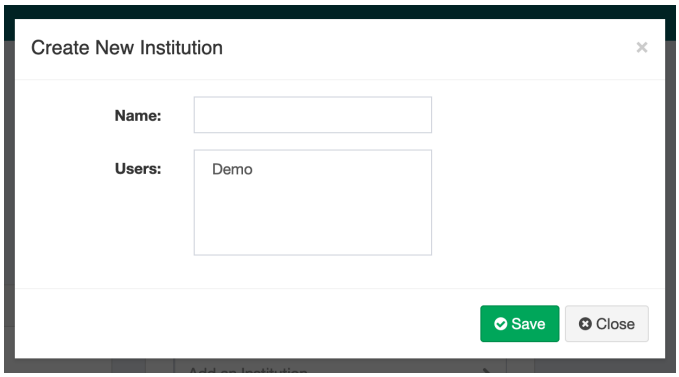

FIGURE 3.3: Modal to create new institution.

3.2 Institutional features

The configuration of a data source is illustrated in Figures 3.4 and 3.5. Firstly is created the data source instance in the institution, and then, the details of this connection are inserted in the modal represented in Figure 3.5.

The connection to the remote server follows a similar flow. This configuration is illustrated in Figures 3.6 and 3.7.

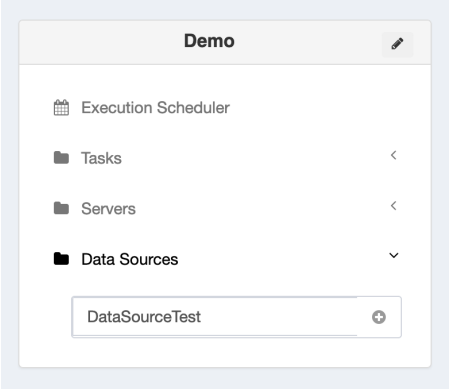

FIGURE 3.4: Creation of connection to a data source (first step).

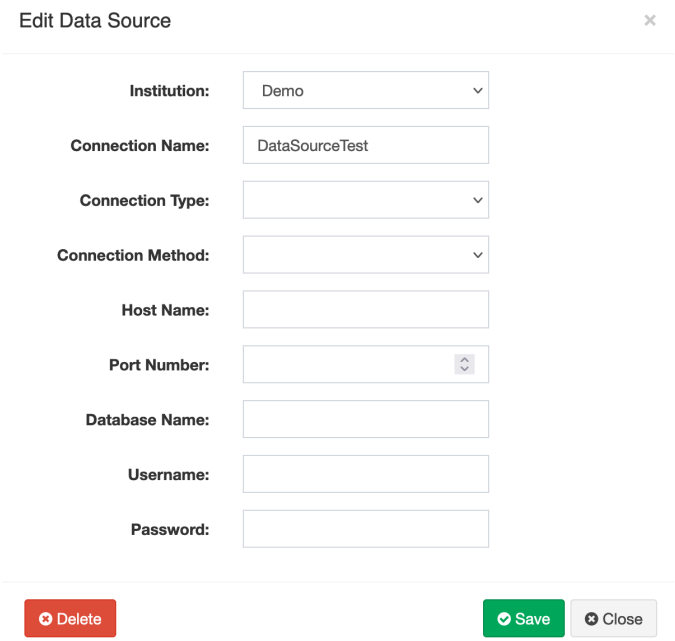

FIGURE 3.5: Modal to insert information regarding the new connection to a data source (second step).

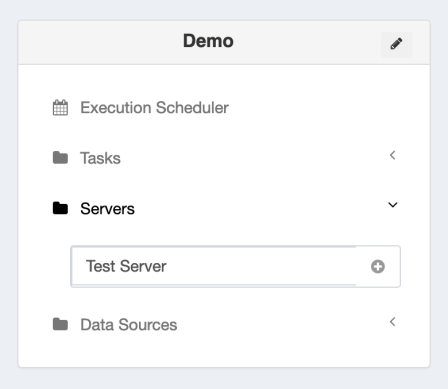

FIGURE 3.6: Creation of connection to a remote server (first step).

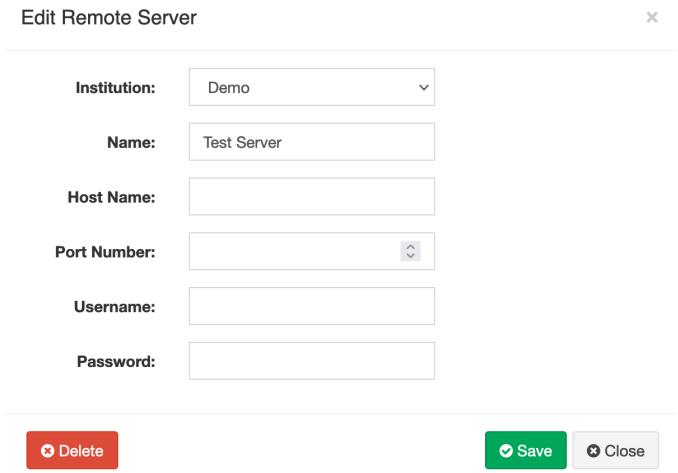

FIGURE 3.7: Modal to insert information regarding the new connection to a remote server (second step).

Finally, the ETL tasks are also created for the institutions. This feature redirects the users to the ETL Task Editor, in which they can design and implement an ETL pipeline using the web interface. The creation of a new task can be done by using the option represented in Figure 3.8.

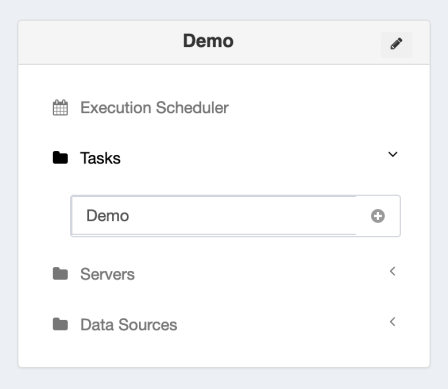

FIGURE 3.8: Creation of a new ETL task (first step).

3.3 ETL Task Editor

The ETL Task Editor allows users to create an ETL pipeline by dragging and dropping ETL components. Figure 3.9 shows an overview of this web editor.

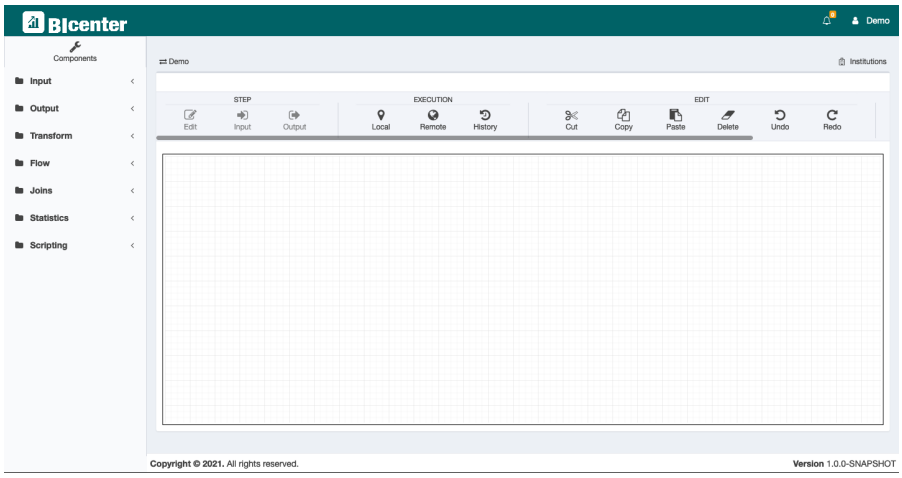

FIGURE 3.9: Overview of the ETL Task Editor.

Figure shows an example of an ETL pipeline implemented in the ETL Task Editor using the four most common ETL components.

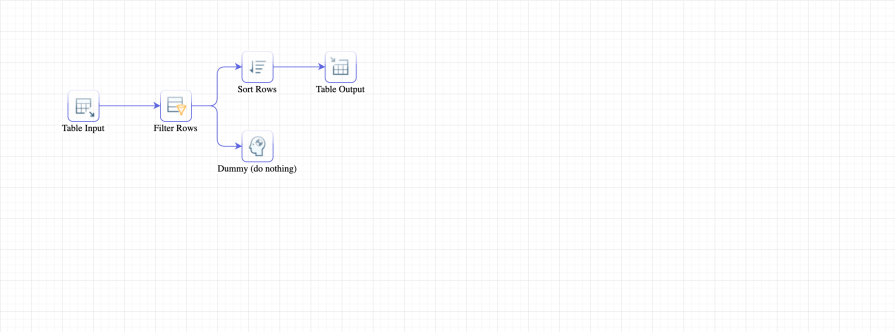

FIGURE 3.10: Simple example of an ETL pipeline represented on this editor

The components available to implement these ETL pipelines are available on the left menu. Part of this menu is represented in Figure , which is expanded the components for input, output and transformations.

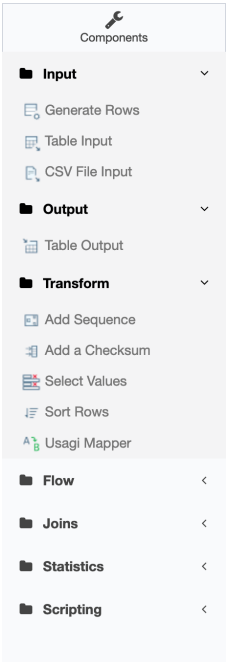

FIGURE 3.11: Overview of part of the ETL components currently deployed in the system.

The navigation bar illustrated in Figure allows the uses to operate over

the ETL Task. This menu is divided into sub-menus: 1) step, which contains the features to configure each ETL component in the pipeline and analyse the input and output of each step; 2) execution, which defined if the ETL task will be executed locally or in a remote server; edit, contain the editing usual features, such as cut, copy, paste, delete, undo and redo; and 4) select, which is related with the drawing features.

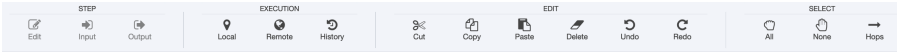

FIGURE 3.12: Main features of the ETL Task Editor.

4

Guidelines for Developers

In this chapter, it is described the main guidelines for contributing to BIcenter.

4.1 Branching Convention

After some consideration, we decided to implement naming conventions in the branches being created. We followed a scheme similar to the one referred at http://stackoverflow.com/a/6065944

Each branch name is composed of the following:

category/*name*

The possible categories are listed below.

Category	Description
bug	Bug fixing
imp	Improvement on already existing features
new	New features being added
wip	Works in progress - Big features that take long to implement and will probably hang there
junk	Throwaway branch created to experimentation

The name should be concise, and directly represent what the branch solves.

Some examples:

bug/issue234

bug/fixeditdb

new/statistics

junk/tryingboostrap3

4.2 Adding an REST API endpoint

1. Create the endpoint at `app` > `controllers`, either on one of the existent JAVA files or a new one if it handles a completely different topic/task, yet to be addressed.
2. Add the respective endpoint URL to:
    - `conf` > `routes`, so that the system knows that endpoint exists;
    - `app` > `controllers` > `Application.java` > `javascriptRoutes()`, so you can use this URL dynamically at JavaScript, regarding frontend code.

Since the structure of this project includes both Frontend and Backend, when you create a REST API endpoint, you'll probably consume it on some frontend feature. Here's how to do that:

1. At `app` > `javascripts` > `services`, there is a collection of JS files that have all the encapsulated logic responsible for communicating with the REST API: one file for each major entity or group of actions. So, in one of these files or a new one, add the function that handles the communication you need with the REST API through your newly developed endpoint.
2. At the JS controller level of your functionality, you call the function specified in the previous step with the necessary arguments and information.

4.2.1 Example

This example assumes that there isn't an endpoint capable of providing information about all the existent institutions. So the purpose is to create such endpoint and provide a function that allows us to consume it on the frontend. Hence, there are 2 phases: the **creation of the REST API**

endpoint and the **creation of the necessary utility to consume it** on the frontend.

4.2.1.1 Creation of REST API endpoint

Because the endpoint we want to create has the purpose of returning the information about all existent institutions, it should be placed at `app` > `controllers` > `InstitutionController.java`, with some code like this:

```
@Security.Authenticated(Secured.class)
@CheckPermission(category = Category.INSTITUTION, needs = {Operation.GET})
public Result getInstitutions(){
    String email = session().get("userEmail");
    List<Institution> institutions = institutionRepository.list(email);

    ObjectMapper mapper = new ObjectMapper();
    SimpleModule module = new SimpleModule();
    module.addSerializer(Institution.class, new InstitutionSerializer());
    module.addSerializer(Task.class, new SimpleTaskSerializer());
    module.addSerializer(Server.class, new ServerSerializer());
    module.addSerializer(DataSource.class, new DataSourceSerializer());
    mapper.registerModule(module);
    Json.setObjectMapper(mapper);

    return ok(Json.toJson(institutions));
}
```

After creating the endpoint, we need to make it discoverable (be available to the outside world). To do that, we add its URL to: 1. `conf` > `routes`:

```
...
GET          /institution/list                    controllers.InstitutionController
...
```

2. `app` > `controllers` > `Application.java` > `javascriptRoutes()`:

```
public Result javascriptRoutes() {
    response().setHeader(Http.HeaderNames.CONTENT_TYPE, "text/javascript");
    return ok(JavaScriptReverseRouter.create("jsRoutes",
        ...,
```

```
        routes.javascript.InstitutionController.getInstitutions(),

        ...
    ));
}
```

4.2.1.2 Creation of utility function to consume the REST endpoint

This second phase explains how to create the referred utility function so we can more easily consume the developed endpoint. Since it's an institution-related method, we will be adding the following utility function to `app` > `javascripts` > `services` > `institution.js`.

```
Institution.getInstitutions = function (callback) {
    jsRoutes.controllers.InstitutionController.getInstitutions().ajax({
    contentType: 'application/json; charset=utf-8',
        success: function (response) {
            if (callback) {
                callback(response);
            }
        },
        error: function (response) {
            console.error('Error in Institution service', response);
        }
    })
 };
```

This consumes the endpoint and, on success calls the specified callback function; otherwise (on fail) displays an error message on your browser's console.

After this, the final part is to consume this utility function, which can also be called a service. A simple example is to use the service on your javascript file that bears the "backend" responsibility of your UI component/view, in a similar way to the following code:

```
Institution.getInstitutions(function (institutions) {
    // do whatever you need with the returned information
});
```

4.3 Adding web view

If you want to create an entirely new base schema (a view), you first need to add a base template name `<name>.scala.html` to `app` > `views`. You also need to make sure that you have an endpoint on JAVA code that handles your request and returns this web view (the "**backend**" process is similar to create a REST API endpoint).

If you just want to create a new UI stage, based on an already defined view, follow these steps: 1. Create an endpoint (and register it) that returns the base view you want to use.

2. At `app` > `assets` > `javascripts`, create the controller and view associated with your new UI stage (the view JS file handles the stuff UI related and the controller handles the stuff Backend related).
3. At `app` > `assets` > `templates`, create a file named `<name>.handlebars` where you'll insert the UI code that will be dynamically added to the main div container of your base web view.
4. After all these files are ready, you need to make sure the framework knows the path to everything you created, so you'll need to add the paths to your view and controller javascript files in the file `app` > `assets` > `javacripts` > `main.js`.
5. Finally, you'll need to add the RegExp handler that will assemble the resources you need (variables and controllers) at `app` > `assets` > `application` > `application.js`.

4.3.1 Example

This example is going to be divided into two big parts: the **creation of a new view from scratch** and the **creation of a UI component** (through handlebars).

Let's take as an example the creation of a dashboard page, given the fact that we need to create a new base-view from scratch.

NOTE: if you just want to create a new UI component, with handlebars, jump to the second part.

4.3.2 Creation of a new (base) view

To create a new view to use as a base to new UI stages, you first need to add the HTML template. So, add your view structure to a file named `home.scala.html`, for the sake of this example, at `app` > `views`:

```
<!DOCTYPE html>

<html>
    <head>
        <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <!-- Tell the browser to be responsive to screen width -->
    <meta content="width=device-width, initial-scale=1, maximum-scale=1, user-scala

    <title>BIcenter</title>

    <!-- Boostrap Select2 -->
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("lib/selec

    <!-- Font Awesome -->
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("lib/font-
    <!-- PNotify -->
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("lib/pnoti
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("lib/pnoti
    <!-- query-builder -->
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("lib/jQuer

    <link rel="shortcut icon" type="image/png" href="@routes.Assets.versioned("imag

    <!--DataTable dependencies -->
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("lib/datat
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("lib/datat

    <!-- jQuery UI -->
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("lib/jquer
```

```
    <!-- Boostrap DateTime-Picker -->
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("lib/boots

    <!-- iCheck -->
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("lib/iChec
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("lib/iChec

    <!-- Main CSS: Bootstrap + AdminLTE + Custom css -->
    <link rel="stylesheet" media="screen" href="@routes.Assets.versioned("styleshee
    </head>
    <body style="height:auto;" class="skin-darkblue-light fixed home-page">
        <script type="text/javascript" src="@routes.Assets.versioned("lib/mxgraph2/
    <script type="text/javascript" src="@routes.Assets.versioned("editor/editor.js"
    <script type="text/javascript" src="@routes.Assets.versioned("editor/graph.js")

    <div class="container">
        <div module="HeaderModule">
            @header()
        </div>

        <div module="MainModule">
            <div controller="HomeController"></div>
        </div>

        <div class="navbar-fixed-bottom">
            @footer()
        </div>
        </div>

        <script type="text/javascript" data-main="@routes.Assets.versioned("javascr
    </body>
</html>
```

After creating the template, you now need to create an endpoint to return this view and register it in the system.

So, to create the endpoint you add a new controller at `app` > `controllers` or add a function to an existent controller. In this example we will create a new controller called `HomeController.java`:

```
public class HomeController extends Controller {
    @Security.Authenticated(Secured.class)
    public Result index() {
        return ok(views.html.home.render());
    }
}
```

And, then, to register it, add to: 1. `conf` > `routes`:

```
...

GET               /home                       controllers.HomeController.index()


...
```

2. `app` > `controllers` > `Application.java` > `javascriptRoutes()`:

```
public Result javascriptRoutes() {
    response().setHeader(Http.HeaderNames.CONTENT_TYPE, "text/javascript");
    return ok(JavaScriptReverseRouter.create("jsRoutes",
        ...,

        routes.javascript.HomeController.index(),

        ...
    ));
}
```

4.3.3 Creation of a UI component (handlebars)

Once the URL for the base view is configured, the next step is to create the functional files that handle frontend and backend of the UI element. So, we will create a folder `home` at `app` > `assets` > `javascripts`. Inside this folder create tho files: `homeController.js` and `homeView.js`.

The `homeController.js` is responsible to handle the backend operations and should follow a structure similar to this:

```
define('HomeController', ['Controller', 'HomeView', 'Router', 'Institution', 'Task'
    const HomeController = function (module) {
        Controller.call(this, module, new HomeView(this));
```

```
    };

    // Inheritance from the superclass
    HomeController.prototype = Object.create(Controller.prototype);
    const _super_ = Controller.prototype;

    HomeController.prototype.initialize = function ($container) {
        _super_.initialize.call(this, $container);

    this.getTasks();
    };

    HomeController.prototype.getTasks = function () {
        const context = this;
    Institution.getInstitutions(function (institutions) {
        context.view.loadInstitutions(institutions);
    });
    };

    return HomeController;
});
```

On the other hand, the `homeView.js`is responsible for frontend interactions and should follow a structure similar to:

```
define('HomeView', ['jquery', 'View'], function ($, View) {
    const HomeView = function (controller) {
        View.call(this, controller, 'home');
    };

    // Inheritance from super class
    HomeView.prototype = Object.create(View.prototype);
    const _super_ = View.prototype;

    HomeView.prototype.initialize = function ($container) {
        _super_.initialize.call(this, $container);
    };

    HomeView.prototype.loadInstitutions = function (institutions) {
        const html = JST['home']({
```

```
        institutions: institutions
        });
        this.$container.html(html);
    this._loadViewComponents();
    };

    return HomeView;
});
```

As can be seen in this last code snippet, we "import" a web (HTML) structure from `'home'`. This refers to the file `home.handlebars` that must be placed at `app` > `assets` > `templates`. Hence, we will create this exact file in the specified location with the following content:

```
<div class="main-div">
    <div class="title">
        <h1><b><i class="fa fa-hospital-o"></i> Dashboard: Resources</b></h1>
    </div>

    <div id="institutions" view-element="institutions" class="row">
        {{#institutions}}
            <div class="institution col-lg-4">
                <div class="panel panel-default">
                    <div class="panel-heading">
                        <h3 class="panel-title"><b>{{name}}</b></h3>
                    </div>
                    <div class="panel-body">
                        <ul class="treeview-menu" data-widget="tree">
                            <!-- Tasks -->
                            <li class="treeview menu">
                                <a name="tasks">
                                    <i class="fa fa-folder"></i> <span>Tasks</span>
                                    <span class="pull-right-container">
                                    <i class="fa fa-angle-left pull-right"></i>
                                </span>
                                </a>
                                <ul class="treeview-menu">
                                    {{#tasks}}
                                        <li>
                                            <a <i class="fa fa-file-text-o"></i>
```

```
                                        <span>{{name}}</span>
                                    </a>
                                </li>
                            {{/tasks}}
                            <li>
                                <form class="sidebar-form">
                                    <div class="input-group">
                                        <input name="taskName" class="form-
                                        <span class="input-group-btn">
                                        <button type="submit" class="btn bt
                                            <i class="fa fa-plus-circle"></
                                        </button>
                                </span>
                                    </div>
                                </form>
                            </li>
                        </ul>
                    </li>

                </ul>
            </div>
        </div>
    </div>
        {{/institutions}}
    </div>
</div>
```

The next step is to tell the system where the newly-created resources are. So, at `app` > `assets` > `javacripts` > `main.js` add the following:

```
requirejs.config({
    baseUrl: '/assets/javascripts',
    paths: {
        ...,

        // Home
    'HomeController': 'home/controllers/homeController',
    'HomeView': 'home/views/homeView',

        ...
```

```
    },
    ...
});

var DEBUG = true;

require(['Application'], function (Application) {
    window.app = new Application();
});
```

Last, but not least, we have to create the RegExp handler, as said earlier, so the system know what controllers and tools to load in the specific UI stage, at app > assets > application > application.js:

```
define('Application', ['jquery', 'Router', 'Module', 'jsRoutes', 'Svg', 'Institutio
    ...

    Application.prototype.initialize = function () {
        var self = this;

    // Configure router
    Router.config({mode: 'history'});

    // Add routes
    Router
            .add(new RegExp(jsRoutes.controllers.login.Login.index().url.substr(1),
            console.log("LOGIN PAGE");
        })
            ...
            .add(new RegExp(jsRoutes.controllers.HomeController.index().url.substr(
            console.log("homepage");

            self.loadControllers('MainModule', ['HomeController']);
        });
    };

    ...

    return Application;
});
```

With this, we will be able to have a web page similar to this image, based on an entirely new HTML template:

New HTML view created using a new UI component following the handlebars pattern.

4.4 Adding new PDI step

Adding and registering a new **Pentaho Data Integration** Step is a straightforward process which has been streamlined by the design of the BICenter project. A list of all available PDI Steps can be found in this link[1].

After selecting the Step to be added, the following is the process required to add said step into the system.

1. Register the new step on the `public > editor > diagrameditor.xml` file.
2. Add a new object to `conf > configuration.json` under the subsection pertaining to the new component's type with the desired component properties.
3. If necessary, add extra functionality to the *submitClick* method on the `app > assets > javascripts > step > stepController.js` or the *applyChanges* method on `app > assets > javascripts > services > step.js`.
4. Create a new class on `app > diSdk > step > parser` with the appropriate name.
5. If the component requires pre-processing or extra logic you can alter the *decodeStep* method on `app > diSdk > step > AbstractStep.java`

4.4.1 Example

The following is an example of how to add the CSVFileInput component. For more information about this component's functioning you can check out this link[2].

[1] https://wiki.pentaho.com/display/EAI/Pentaho+Data+Integration+Steps
[2] https://wiki.pentaho.com/display/EAI/CSV+File+Input

1. Firstly, we have to add the following line to `public > editor > diagrameditor.xml`

```
<add as="CSV File Input" template="CSVInput" icon="/assets/images/editor/rectangle.
```

2. Next we'll have to register the component's properties on the `conf > configuration.json` file. As the CSV File Input step is an Input type component, we'll be registering it under the Input Components. We'll be naming this object the same name we used on the `template` field on the prior step. Note that the `shortName` field should have a name that goes in accordance with the naming specified by the Pentaho Kettle library (in our specific case, this can be seen in this link[3]). Under the `componentProperties` array we can specify the multiple inputs and fields of our component.

```
{
        "name": "CSVInput",
        "description": "CSV File Input",
        "shortName": "csvInput",
        "componentProperties": [
          {
            "name": "Step Name",
            "shortName": "stepName",
            "type": "input"
          },
          {
            "name": "File Name",
            "shortName": "fileName",
            "type": "fileinput"
          },
          {
            "name": "Delimiter",
            "shortName": "delimiter",
            "type": "input"
          },
          {
            "name": "Enclosure",
```

[3] https://javadoc.pentaho.com/kettle/org/pentaho/di/trans/steps/csvinput/CsvInputMeta.html

```
            "shortName": "enclosure",
            "type": "input"
          },
          {
            "name": "NIO Buffer Size",
            "shortName": "bufferSize",
            "type": "number"
          },
          {
            "name": "File Encoding",
            "shortName": "encoding",
            "type": "select",
            "componentMetadatas": [
              {
                "value": "UTF-8",
                "name": "UTF-8"
              },
              {
                "value": "ANSI",
                "name": "ANSI"
              }
            ]
          },
          {
            "name": "Lazy Conversion?",
            "shortName": "lazyConversionActive",
            "type": "checkbox"
          }
        ]
      }
}
```

3. As BICenter is already prepared to receive files and all of our CSVInput's parameters, we can skip this step.

4. Now we have to create a new class on the `app > diSdk > step > parser` directory. This can simply be done by copying any of the other classes already present in the folder. Don't worry about the lack of logic in this class, as this serves as a mere extension of the AbstractStep class, which itself contains all

logic needed to communicate with the PDI, used in order to allow our component to be detected and processed.

```
package diSdk.step.parser;

import diSdk.step.AbstractStep;
import models.Step;
import org.pentaho.di.trans.step.StepMetaInterface;
import org.w3c.dom.Element;

public class CSVInput extends AbstractStep {
    @Override
    public void decode(StepMetaInterface stepMetaInterface, Step step) throws Excep

    }

    @Override
    public Element encode(StepMetaInterface stepMetaInterface) throws Exception {
        return null;
    }
}
```

5. As we want our system to automatically detect the CSV's fields automatically without the users having to manually introduce them themselves, we have to add some logic to the *decodeStep* method on `app > diSdk > step > AbstractStep.java` which will do an initial read of the file in order to extrapolate it's columns' names. This can be done by creating a new method on this class and adding it to the *decodeStep* method, or by injecting the logic directly into the function.

```
// If dealing with CSVFileInput get the input fields and define them
                        if (shortName.equals("InputFields")) {
                            if (fileName == null) {
                                Optional<StepProperty> fileNameStepProperty = stepP
                                        .filter(stepProperty -> stepProperty.getCom
                                        .findFirst();

                                if (!fileNameStepProperty.isPresent())
                                    continue;
```

```
    fileName = fileNameStepProperty.get().getValue();
}

if (delimiter == null) {
    Optional<StepProperty> delimiterStepProperty = stepI
            .filter(stepProperty -> stepProperty.getComp
            .findFirst();

    if (!delimiterStepProperty.isPresent())
        continue;
    delimiter = delimiterStepProperty.get().getValue();
}

try {
    BufferedReader br = new BufferedReader(new FileReade
    String header = br.readLine();

    String[] fields = new String[0];
    if (header != null) {
        fields = header.split(delimiter);
    }

    TextFileInputField[] value = new TextFileInputField
    for (int i = 0; i < fields.length; i++) {
        String field = fields[i];
        value[i] = new TextFileInputField();
        value[i].setName(field);
        System.out.println(field);
    }

    // Invoke the current method with the StepProperty
    invokeMethod(stepMetaInterface, method, value, datal

} catch (FileNotFoundException e) {

}
```