CONTENTS

[1 Introduction 2](#_Toc422998311)

[2 Design requirements 2](#_Toc422998312)

[3 DSS setup 3](#_Toc422998313)

[4 Structure of the DSS environment application 3](#_Toc422998314)

[5 Database 5](#_Toc422998315)

[5.1 Structure 5](#_Toc422998316)

[5.2 How to create a table in the database 6](#_Toc422998317)

[5.3 Examples of how to work with the database 6](#_Toc422998318)

[5.3.1 Insert 6](#_Toc422998319)

[5.3.2 Update 6](#_Toc422998320)

[5.3.3 Delete 7](#_Toc422998321)

[6 Front-end sections 7](#_Toc422998322)

[6.1.1 List of buildings 7](#_Toc422998323)

[6.1.2 Building X 7](#_Toc422998324)

[6.1.3 Historical & Forecasted data 8](#_Toc422998325)

[6.1.4 PV Action Plan 8](#_Toc422998326)

[6.1.5 Administration: list of buildings 9](#_Toc422998327)

[6.1.6 Administration: Building partition 9](#_Toc422998328)

[6.1.7 Administration: Sensors 10](#_Toc422998329)

[6.1.8 Administration: Action Plan list 10](#_Toc422998330)

[6.1.9 Administration: Mapping 11](#_Toc422998331)

[7 Processes 11](#_Toc422998332)

[7.1.1 To generate a new prediction 11](#_Toc422998333)

[7.1.2 Query the historical and forecasted data for a particular date 12](#_Toc422998334)

[8 References and links 12](#_Toc422998335)

# Introduction

This report describes the work done in Task 3.4 “Front-end environments”. In particular, the report descrbies the design and development of the DSS environemnts.

The DSS environments have been implemented as a PHP application. There are basically two: one for the end-user and a second one for the administrator user. Both environments are accessible through this URL:

<http://arcdev.housing.salle.url.edu/optimus/optimus/web/app.php/login>

username: admin

password: admin

After logging in, the user can go to the end-user environment by clicking the *“Analyze”* button or to the administration environment by clicking the *“Admin”* button.

The DSS environments have been developed as a Symfony2 application (<https://symfony.com>).

# Design requirements

The DSS environments have been designed taking into account the following:

* The DSS should be open and flexible enough to be expanded in the future with new cities, buildings, sensors, action plans, languages….
* The front-end should be multilingual (English, Spanish, Catalan, Italian….)
* The monitored data is stored in a Virtuoso server by means of the Semantic Framework (Describe in D3.1). That is, each variable (e.g., outdoor temperature) is sent to the DSS through a Ztreamy stream in RDF.
* The predictions are carried out each day at night for each variable needed by the action plans. The predictions have a time horizon of 7 days. This way, each day a prediction for the following 7 days is carried out. The predictions are not stored in Virtuoso (used for real monitored data), but in the MariaDB database because it is variability. The database has been designed to let the user to create a prediction in the past, for example for testing the system.
* The action plans are pre-calculated just after the predictions have been carried out. This way the users visualize the output of the action plans and the monitored data. The outputs of the action plans are stored in MariaDB database.
* Each instance of the DSS environments models one City. This way, all the sensors and buildings setup in the DSS refer to the city. We will install 3 different instances of the DSS environments, they will share the virtuoso server (using different named graphs).
* The process of setup the DSS is the following:

1. Create a new building.
2. Create the building partitions according to the reality.
3. Add sensors which should be accesible through the Semantic Framework.
4. Enable the action plans to be used in the building.
5. Mapping the sensors to the action plans inputs.

* The main elements of the DSS are: Sensors, buildings, partitions, predictions, action plans, and action plan outputs.
* A building has partitions, sensors, and action plans.
* An action plan has a list of sensors that provide data for the calculations and the results of the calculations.
* An action plan has a predefined set of input variables. For example the PV plan needs: energy production, energy consumption and energy prices.
* An action plans can be applied in any building partition. It is a matter of mapping sensors to the input variables needed for the action plan in a specific partition.
* The outputs of the calculations of an action plan are stored in the database in a particular tables for each action plans. (see database structure).
* A sensor is a device that monitors something. It might be related to a physical device. The sensors send data through the Semantic Framework to the DSS in RDF format. The sensors are identified by a particular URI[[1]](#footnote-1).
* The prediction models (developed in T3.2) get data from the sensors –through the Semantic Framework– and forecast them for the next 7 days. The predicted data is stored in the database. A variable for the same day can have multiple predicted values.

# DSS setup

In order to setup a DSS instance, the following steps have to be carried out using the administration environment.

1. Create a new building.
2. Create the building partitions according to the building reality.
3. Add the sensors which should already been installed through the Semantic Framework.
4. Enable the action plans to be used in the building.
5. Mapping the sensors to the action plans inputs.

# Structure of the DSS environment application

The DSS envirionments has been developed as a Synfony application. The folder structure is given by Synfony framework. It can summaried in the the following components:

* **Controlers**: basic functionality of the environemntes, is the input entrance of the HTML petitions. Usually each controlers manages a web page (login, users, actions plans…)
* **Entities**: contains the mapping between the database and the synfony application. Each table has a entity with the getters and setters.
* **Forms**:
* **Repostiory**:
* **Views**: it is the front-end of the application. Each page is a view.
* **Services**: they are the main scripts that handles the data and process it to send it to the views. The action plans are coded as services.

The complete structure of the DSS application is the following:

/app

/config

- **config.yml:** *Configuration file of the application (users, parameters, languages...)*

- **routing.yml:** *Bundle registration*

- **parameters.yml**: *Database configuration*

/Resources

/views/

**- base.html.twig**: *Basic HTML layout, it contains the header of the DSS front-end*

/src

/Optimus/OptimusBundle/

/Controller (All controllers)

- **AdminActionPlanController**: *functions for the administration of the action plans*

- **APSensorsController**: *functions for mapping sensors to action plan*

- **BuildingController**: f*unctions manage a building*

- **InitController**: *methods for initial web pages (init, login…)*

- **LoginController**: *Methods for user management.*

- **PartitionController**: *Methods for partition management*

- **PredictionController**: *Methods for creating the charts of the historic & prediction page*

- **SensorController**: Methods for managing sensors

- **PVActionPlanController**: *Methods for the PV Action Plan*

- **SwitchActionPlanController**: *Methods for the preheating/precooling Action Plan*

/Entity

- *a list of entities for each table of the database*

/Form

- *a list of forms for each entity*

/Repository

- *Advanced methods for handling the database access*

/Resources

/config/

- **routing.yml**: *route declaration*

- **services.yml**: *service declaration*

/translations/

- *files with the different translations*

/views/

- *views of the application*

/Servicios

- **GestorOntologia**: *Methods for retrieving data from Virtuoso*

- **GestorInvoke**: *Methods for invoke RapidminerServer (Prediction models)*

- **GestorCalculos**: *Methods for invoking the predicition models and the action plans*

- **GestorDataCapturing**: *Methods for obtaining the data needed for creating charts*

- **GestorEventos**: *Methods for logging the events (user actions)*

- **GestorAPPV**: *Methods for the PV action plan*

- **GestorAPPreheating**: *Methods for the Pre-heating action plan*

- **GestorAPAdaptative**: *Methods for the Adaptative comfort action plan*

- **GestorAPTSV**: *Methods for the TSV action plan*

/web/

/bundles/

/optimus/

/css

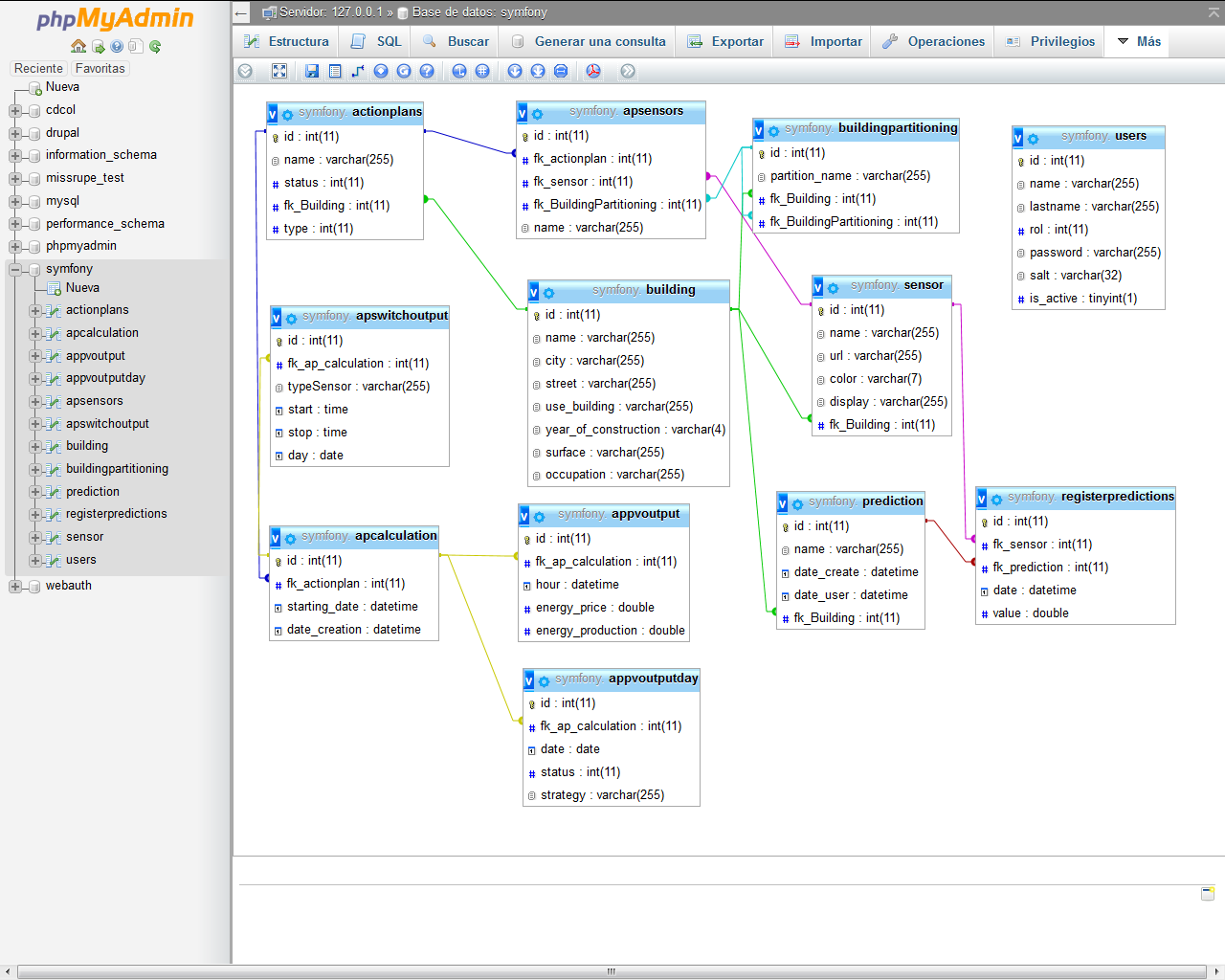
/js

/img

/files

# Database

## Structure



* **Building**: Building basic properties.
* **ActionPlans**: List of action plans. The action plans are pre-defined in Optimus. This table is used to enable/disable them (status).
* **BuildingPartitioning**: Partitions of the buildings. A partition can have other partitions. It follows a tree structure.
* **Prediction**: Predictions for a building. It has two dates, the date of the time period to create the prediction and the specific date and time when the predicition was generated. This way a prediction can be carried out in the past for testing purposes.
* **RegisterPredictions**: For each prediction, the values are stored for all active sensors.
* **Sensor**: List of sensors, by means of the URI the Virtuoso server can be queried to check if the sensor is sending data or not.
* **Users**: List of users.
* **APCalculation**: Record of the action plans calculations for a specific date.
* **APSensors**: Mappings between sensors and partitions for a particular action plans.
* **APPVOutput**: Specific table of the PV action plan. It stores the values needed for the calculations.
* **APPVOutputDay**: Specific table of the PV action plan. It stores the inputs of the user for each day (strategy and status: accept, decline…).
* **APSwitchOutput**: Specific table for preheating action plan. It stores the values needed for the calculations.

## How to create a table in the database

Carry out the following steps to create a table using Synfony procedures.

1. Open the php console and go to the path:

C:/xampp/htdocs/optimus

2. Create a new entity (table) using the following statement:

php app/console doctrine:generate:entity

3. Generate the getters and setters for the entity

php app/console doctrine:generate:entities Optimus/OptimusBundle/Entity/Product

4. Modify the table

php app/console doctrine:schema:update --force

## Examples of how to work with the database

### Insert

$sensor = new Sensor (); //constructor

$sensor->setName ($name);

$sensor->setUrl ($url);

$sensor->setColor ($color);

$sensor->setDisplay ($display);

$em = $this->getDoctrine()->getManager();

$em->persist ($sensor);

$em->flush ();

### Update

$sensor =$em->getRepository ('OptimusOptimusBundle:Sensor')->findBy(array("id"=>$idSensor));

If ($sensor)

{

$sensor->setName ($name);

$em->persist ($sensor);

$em->flush ();

}

### Delete

$sensor=$em->getRepository('OptimusOptimusBundle:Sensor')->find($idSensor);

If ($sensor)

{

$em->remove ($sensor);

$em->flush ();

}

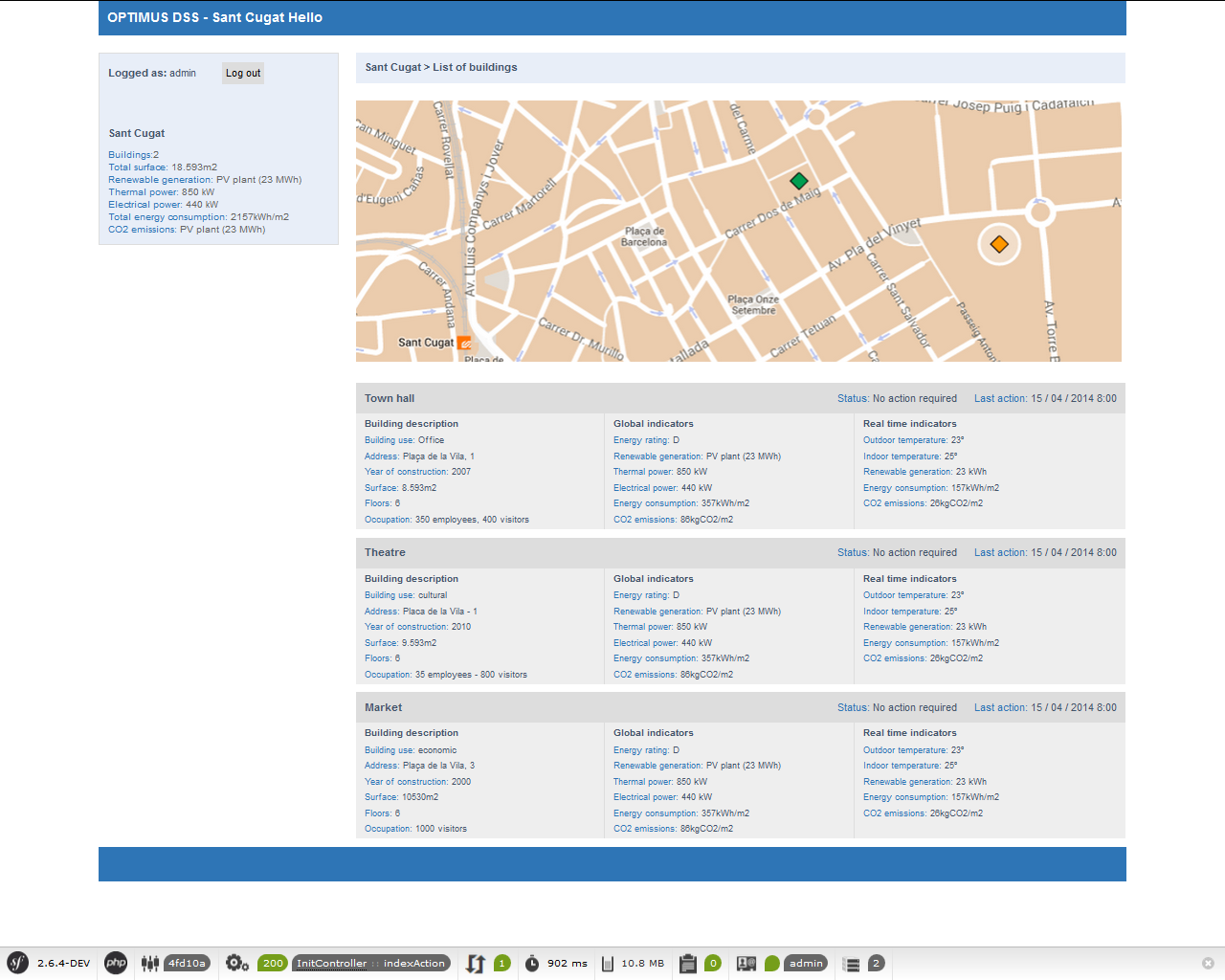
# Front-end sections

### List of buildings

List of the buildings of a city

**Controllers**: InitController

**View**: Building/selectBuilding

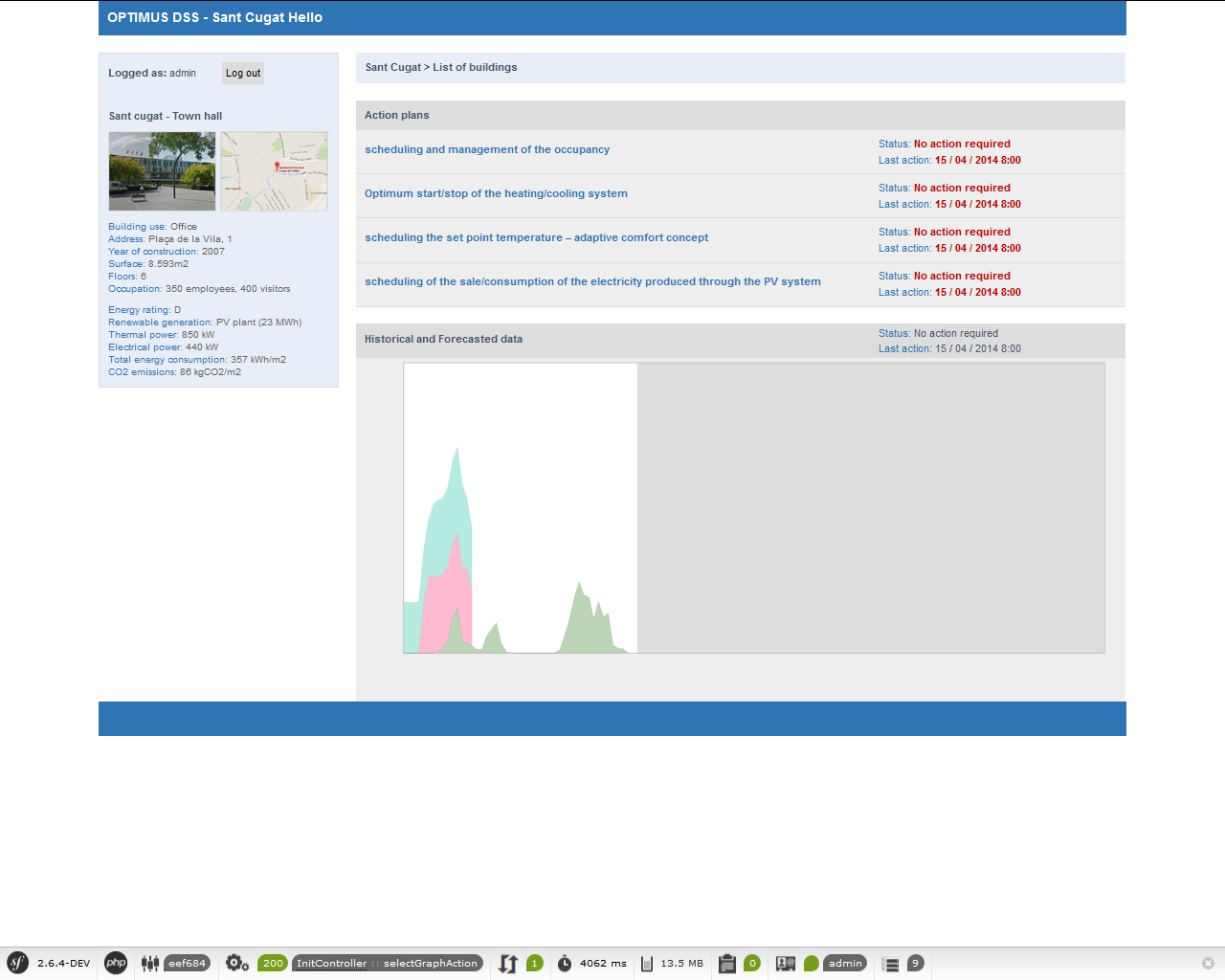


### Building X

A building selected. Its action plans and summary chart is displayed.

**Controller**: InitController

**View**: Building/selectGraph

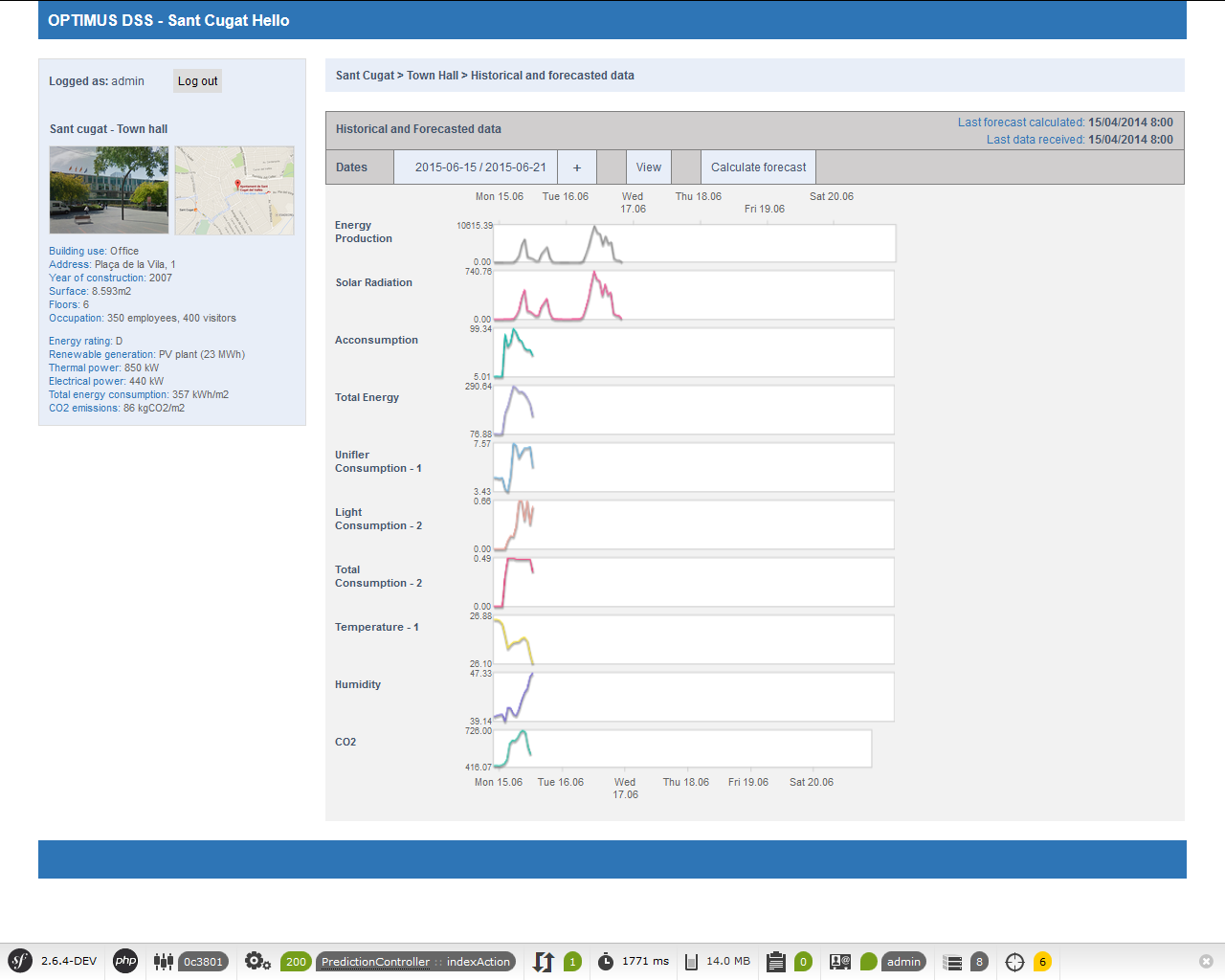


### Historical & Forecasted data

Visualization of the charts of the data (historical and predicted) from the sensors of a building.

**Controller**: PredictionController

**View**: Prediction/graphPredictions

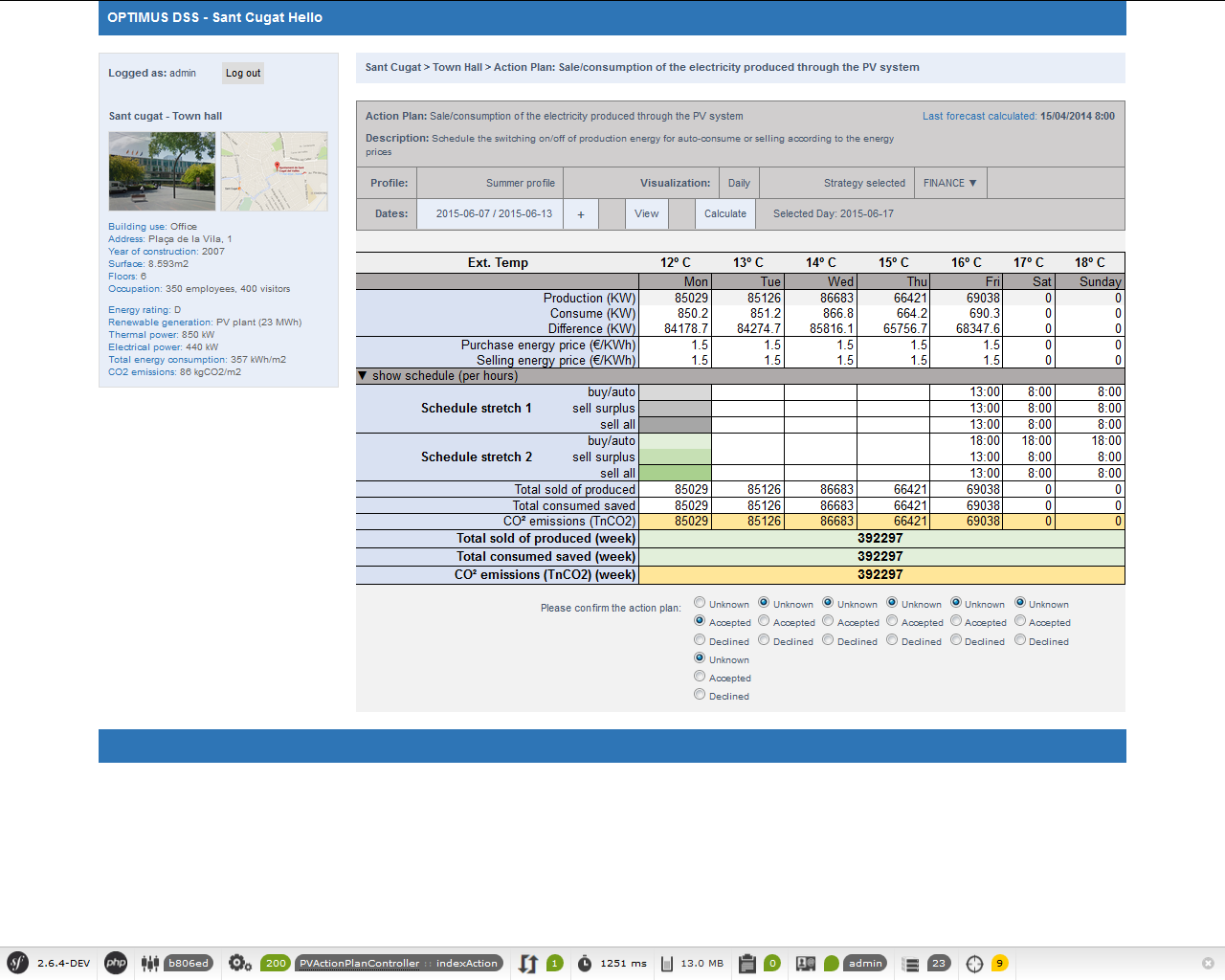


### PV Action Plan

Interface of the PV Action Plan

**Controller**: PVActionPlanController

**View**: PVActionPlan/ PVActionPlan

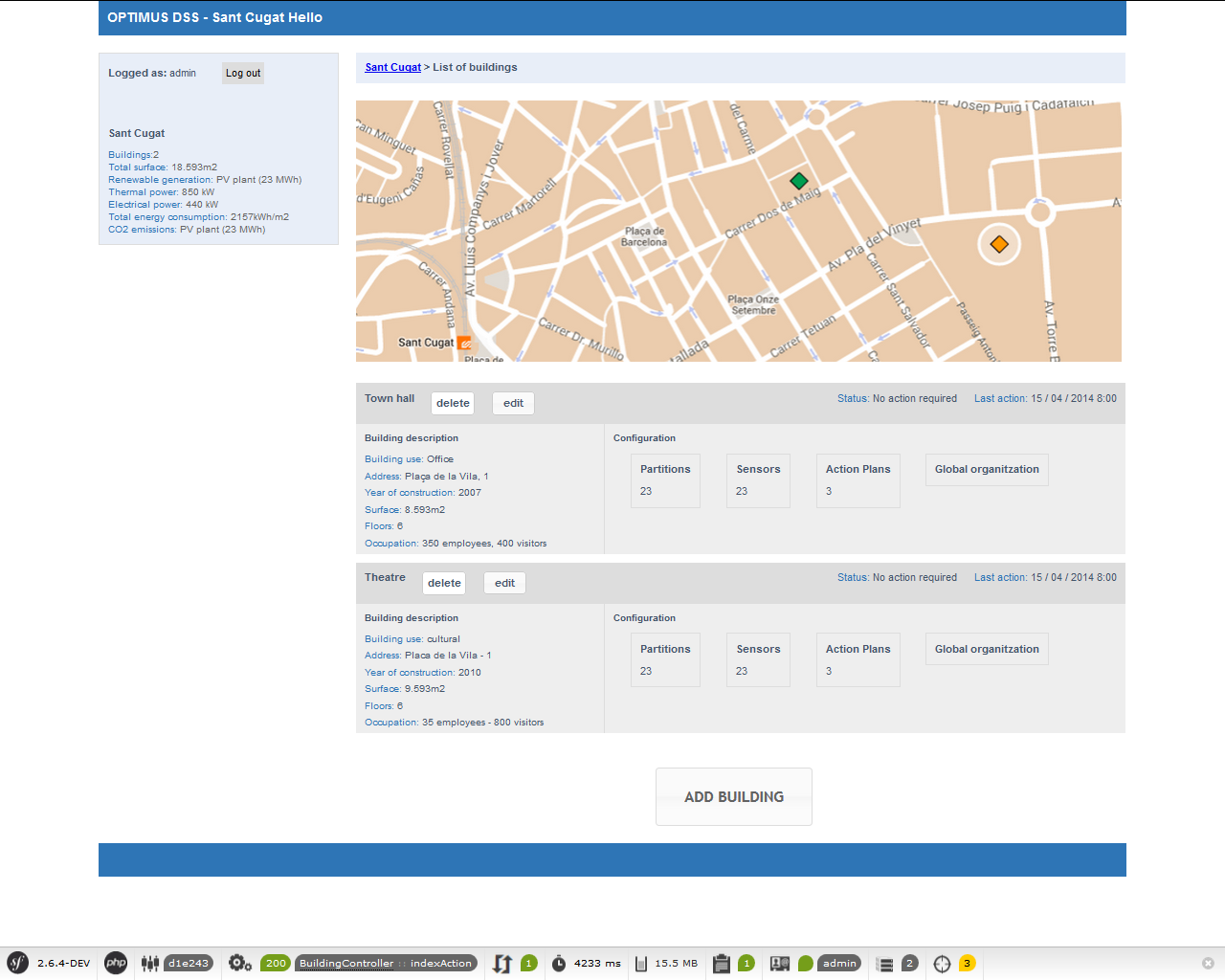


### Administration: list of buildings

Interface for managing buildings: create, modify, remove

**Controller**: BuildingController

**View**: Admin/adminBuildings, Building/new, Building/edit

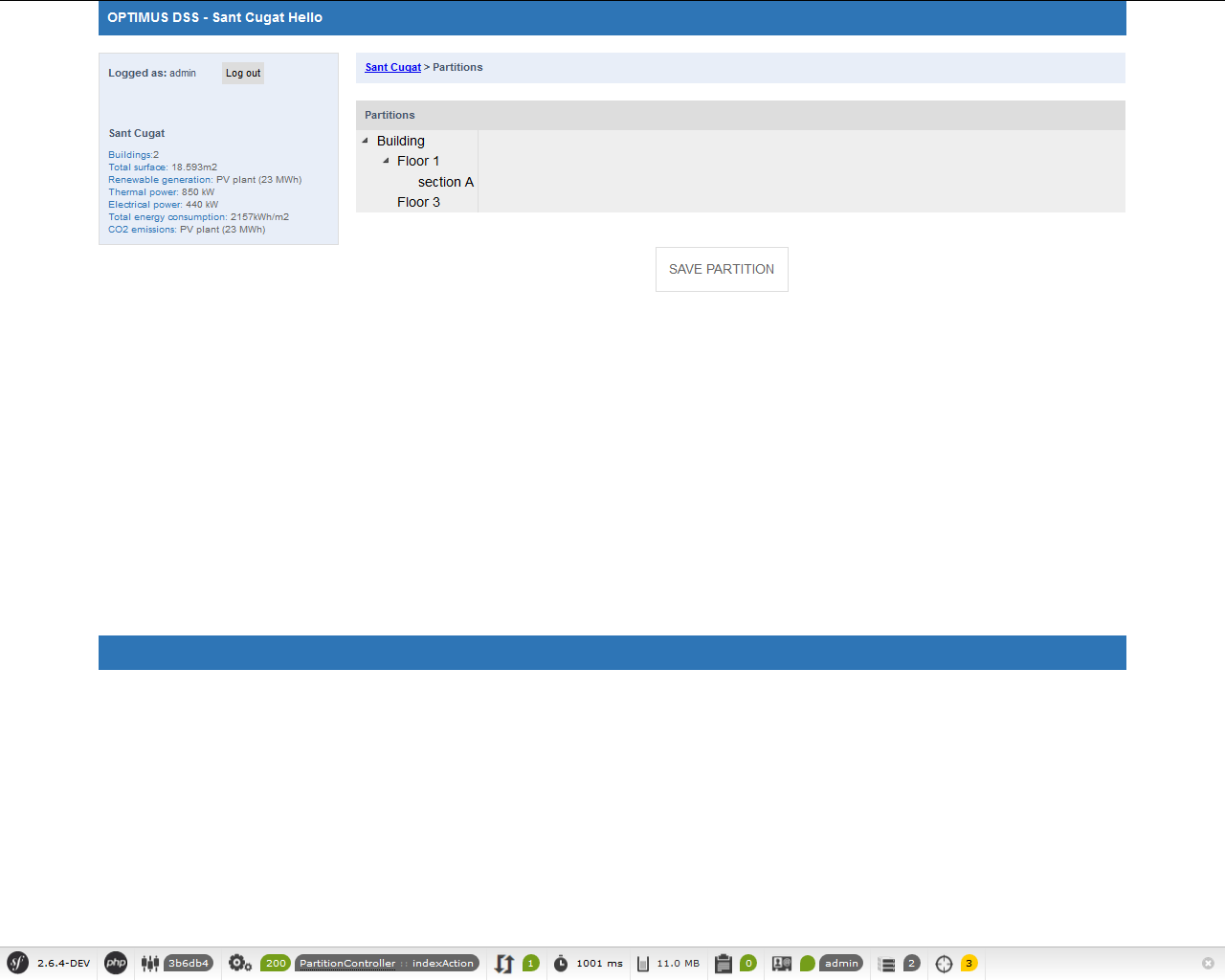


### Administration: Building partition

To create, modify, remove building partitions

**Controller**: - PartitionController

**View**: -Admin/ adminPartitions

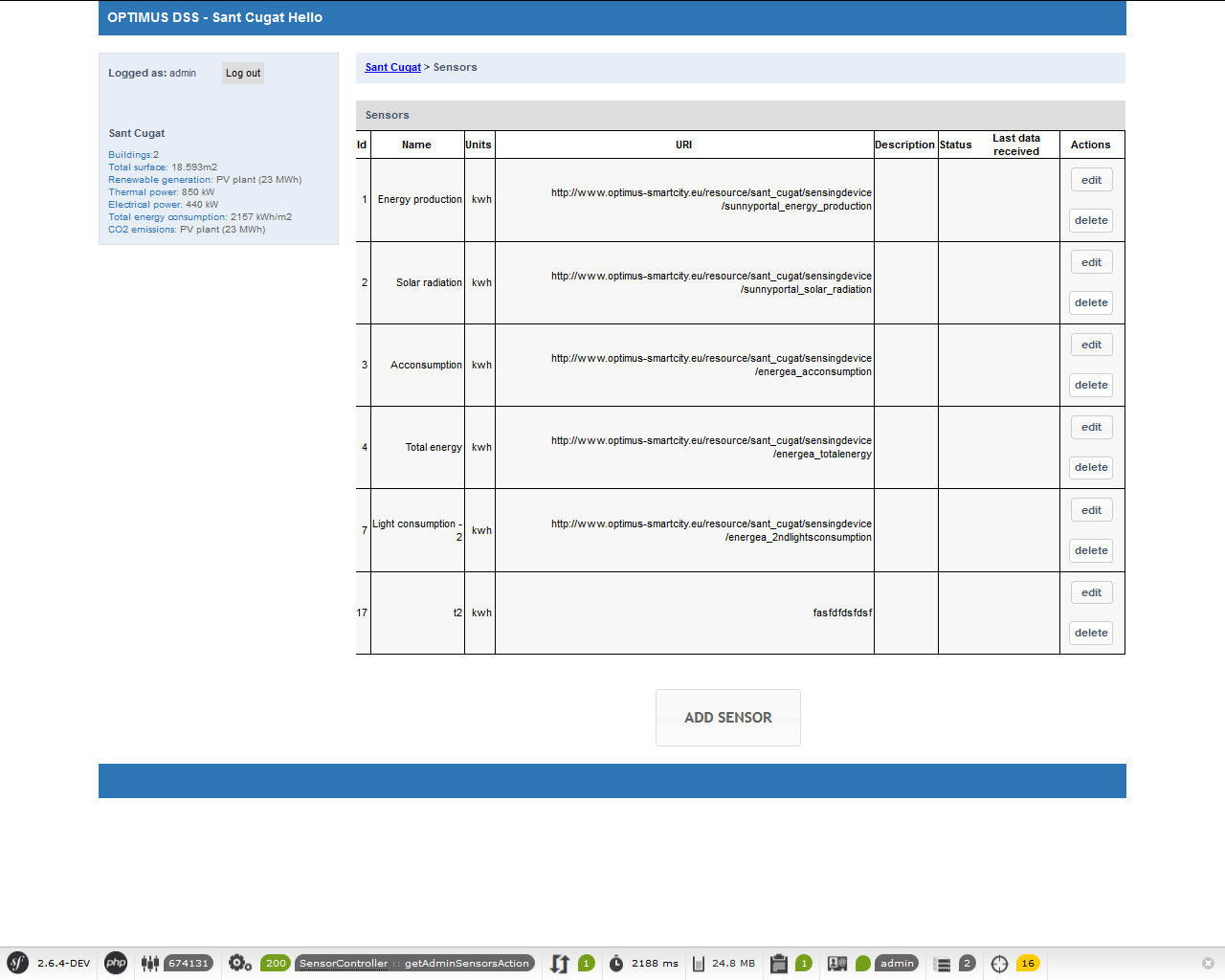


### Administration: Sensors

List of sensors. Interface for create and remove sensors. The sensors should be already configured in the Semantic Framework..

**Controller**: -SensorController

**Views**: Admin/adminSensors, Sensor/new, Sensor/edit

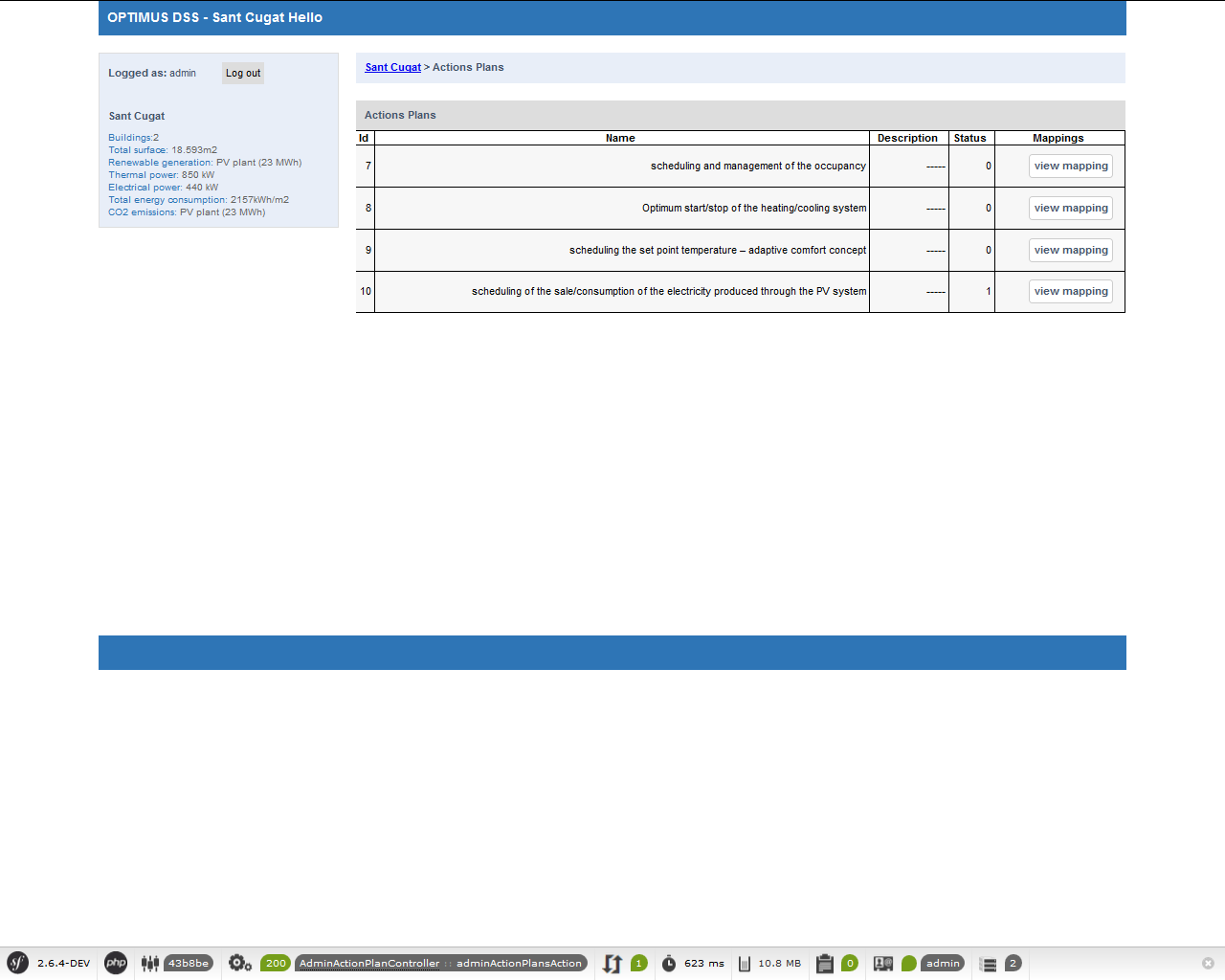


### Administration: Action Plan list

To enable and disable action plans for a specific building.

**Controller**: AdminActionPlanController

**View**: - Admin/adminActionPlans

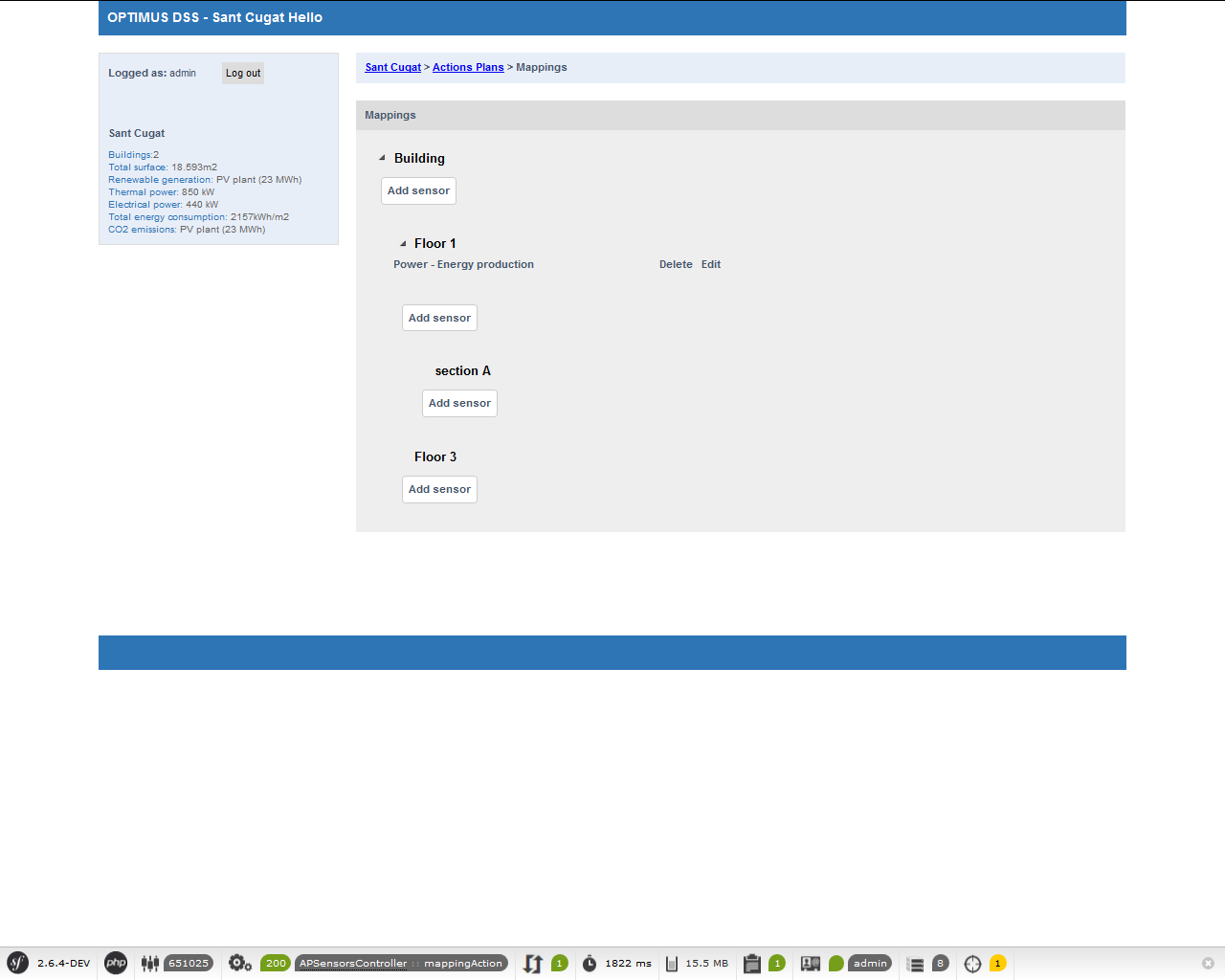


### Administration: Mapping

Mapping between the sensors and partitions in the framework of a specific action plan.

**Controller**: APSensorsController

**View**: APSensors/mapping, APSensors/edit



# Processes

### To generate a new prediction

The “PVActionPlanController” controller receives an input date (usually each day) and invokes the “GestorCalculos” service which needs the input date and the building ID.

The “GestorCalculos” service generates a new prediction and then invokes the Action Plan calculations.

For generating a new prediction the following methods are used:

* **createPrediction**: Inserts a row in the “predictions” table.
* **loadXml**: Invokes the “GestorInvoke” service which is responsible of the communication with RapidMiner Server processes. Those processes returns the predictions in XML format.
* **insertPredictions**: Inserts in the “register\_predictions” table the data after parsing the XML file.

Once all the predictions have been carried out, the calculation of the action plans is invoked.

* **getSensorsByActionPlan**: Obtains the sensors required by the action plan.
* **createNewCalculation**: Inserts a row for each action plan in the table “ap\_calculations”.
* **insertOutputsActionPlan**: Each action plan is calculated using the methods of the corresponding service.

### Query the historical and forecasted data for a particular date

The service “GestorDataCapturing” receives a date and the building ID to retrieve the data for a whole week.

* **getVariablesActives**: Returns the sensors for a particular building.
* **getDataVariables**: Controls if the data for a particular day should be historical data or forecasted. If there are historical data it will be returned instead of the forecasted.
* **getDataVariableHistorical**: Obtains the historical data from Virtuoso server using the service “GestorOntologia”.
* **getDataVariablePrediction**: Obtains the predicted data previously stored in the database.

# References and links

Framework Symfony2 (required min PHP 5.3.8): <https://symfony.com>

Install Symfony2: <https://symfony.com/doc/current/book/installation.html>

Symfony2 guide online (spanish): <http://librosweb.es/libro/symfony_2_4>

ORM Doctrine: <http://www.doctrine-project.org>

Doctrine-Association Mapping (relations): <http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/association-mapping.html>

Template system for the views: Twig <http://twig.sensiolabs.org>

Xampp Web Server (Apache + MySQL + PHP + Perl): <https://www.apachefriends.org/es/index.html>

1. For example: http://www.optimus-smartcity.eu/resource/sant\_cugat/sensingdevice/sunnyportal\_energy\_production [↑](#footnote-ref-1)