Table of contents

1. Introduction 1
2. Server-side Atlas API 2
   - 2.1. File hierarchy 2
   - 2.2. Naming convention 2
   - 2.3. ModuleInterface 3
   - 2.4. Atlas singleton 3
   - 2.5. Other classes 4
   - 2.6. Example plug-in code 5
3. Browser-side Atlas API 7
   - 3.1. Attributes and methods 7
   - 3.2. Common functions 7
   - 3.3. Square object 8
   - 3.4. Hooks 8
   - 3.5. Example plug-in code 9

1. Introduction

Below You'll find a brief description of the Atlas API available to additional module coders as well as a brief tutorial on making your own plug-in compatible with the Atlas platform. The description is divided into sections describing the server-side capabilities and browser-side capabilities given to plug-in developers.

2. Server-side Atlas API

The language of choice for Atlas is PHP5 along with Zend Framework. Atlas is developed using Model-View-Controller (MVC) paradigm, separating data, display and business logic of the application.

A minimum amount of files needed to create a plug-in is 2. One for the server side logic, and one for browser scripting. Below You'll find information on the server side of things.

2.1.File hierarchy

The Atlas project has the following directory structure:

- application – contains all application code
  - configs – contains configuration files
  - controllers – contains all controller files of Atlas core
  - interfaces – contains interface definitions
  - layouts – contains layout files
  - models – contains all model files of Atlas core
  - modules – contains all plug-ins and a plug-in autoloader class
  - views – contains all view scripts of the Atlas core
  - Bootstrap.php – the bootstrapping file of Atlas
- css – contains all CSS style sheets of the Atlas core
- img – contains all image files of tha Atlas core
- js – contains all JavaScript files of the Atlas core
- library – contains Zend Framework
- index.php – the main application file.

The above structure is derived from a basic Zend Framework project layout.

All plug-in files should be located in application/modules/$MODULENAME/ directory (e.g. Predict module is located in application/modules/Predict/)

2.2.Naming convention

All classes added to the project have to follow the rules given here to be visible by the module autoloader class. It is not obligatory apart from the main plug-in file, but it saves on including the class files by hand.

All underscores "_" in a class name mean a directory, keep that in mind when naming your classes. It is possible to have an elaborate directory structure in you module by naming the classes appropriately.

All class names must begin with "Module_" followed by $MODULENAME. This can be followed by the class name or additional directory prefixes (e.g. the Module_Predict_Predictions class is located in application/modules/Predict/Predictions.php).

The main module class must be named Module_$MODULENAME_$MODULENAMEModule (Module_Predict_PredictModule for the Predict plug-in) and must be located in application/modules/$MODULENAME/$MODULENAMEModule.php.

If your module requires a CSS file, put it in your module folder and name it with your module's name ($MODULENAME.css). It will be linked to automatically.

2.3.ModuleInterface

ModuleInterface must be implemented by the plug-in main class. It is located in application/interfaces/ModuleInterface.php and consists of the following methods:

- GetSquareData (Application_Model_Square $aSquare)
  - Called every time square data is required
  - Must return an associative array with square parameters that will be available in JavaScript, as object properties.
- SquareCreated (Application_Model_Square $aSquare, SimpleXMLElement $aXML)
  - Called when a square is created, when Atlas is open for the first time for a particular town. If You need to initialize Your module in any way for a new map, this is the place to do it.
- HandleForwardedRequest()
  - Called when browser calls forward-ajax action with a proper mod parameter (mod=$MODULENAME).
  - If it returns an array, the data will be available in the returned JSON object.
- HasOwnTab ()
  - NOT IMPLEMENTED YET
- GetTabName ()
  - NOT IMPLEMENTED YET
- GetTabContent ()
  - NOT IMPLEMENTED YET
- PreMapHTML ()
  - Called in the view script, before main map HTML is printed
  - Should return a string containing HTML to be injected.
- PostMapHTML ()
  - Called in the view script, after main map HTML is printed
  - Should return a string containing HTML to be injected.
- GetFormElements ()
  - Called when square update form is being prepared, during page refresh.
  - Should return an array of Zend_Form_Element objects that will be added to the update form.
- UpdateFormSubmitted (Zend_Form $aForm)
  - Called after user submits the update form.
  - Should return an array containing coordinates of squares that were changes and need updating. Each square should be an associative array with "x" and "y" keys [e.g. array("x"=>$x, "y"=>$y)].
  - Each zone returned will be updated by a call to GetSquareData() and new data will be available in JavaScript (see: Atlas.SquareDataRecieved() in JavaScript section).
- XMLUpdated (SimpleXMLElement $aXML)
  - Called each time the XML feed is updated, either on page load or by an AJAX call.

- Should return an array containing coordinates of squares that were changes and need updating. Each square should be an associative array with “x” and “y” keys [e.g. array(“x”=>$x, “y”=>$y)].
- Each zone returned will be updated by a call to GetSquareData() and new data will be available in JavaScript (see: Atlas.SquareDataRecieved() in JavaScript section).

2.4.Atlas singleton

A lot of information is available through an Application_Model_Atlas, a singleton class available through Application_Model_Atlas:: getInstance() call. It serves as an information repository as well as a common interface for session data management and data caching (which will be implemented soon to provide performance boost).

The following methods and attributes are available:

- methods
  - getInstance() – returns the Atlas singleton instance
  - getSquare($x, $y) – returns an Application_Model_Square representing a zone at $x,$y coordinates or null if the coordinates are wrong.
  - populateIndex() – a helper method that populates creates all map squares in one call, reducing needed SQL queries. Use this if You need to iterate over all squares before calling getSquare for each of them.
  - setXML($aXML) – sets the XML feed contents ($aXML is a string) to Atlas.
- Attributes
  - ->session – this is an instance of Zend_Session_Namespace (see: http://framework.zend.com/manual/en/zend.session.basic_usage.html), an interface for storing data in a session. It’s simple to use, just save data as it’s property (e.g. $atlas->session->myData = 69) and it will be available in all following calls:

```
if ( is_null($lAtlas->session->XML) ) {
      $lAtlas->session->XML = $this->getFeedData();
}
$lAtlas->setXML( $lAtlas->session->XML );
```

  - ->cache – NOT IMPLEMENTED YET
  - ->squares – map square index (see: Application_Model_Square), a two dimensional array indexed with x and y coordinates ($atlas->squares[$x][$y] will give you the Application_Model_Square for given parameters or null if they are wrong). It should only be accessed directly if You really know what You’re doing, otherwise use getSquare($x, $y) as it will create the needed object if it’s not ready yet, while a direct call to the array may return null.
  - ->XML – the contents of the XML feed as a SimpleXMLElement instance. All feed contents can be accessed through it. See SimpleXML documentation (http://www.php.net/manual/en/simplexml.examples-basic.php) for details.

2.5.Other classes

The other classes You’ll come in contact with are:

- Application_Model_Square – the class represents a map square. It has the following properties and methods:
  - ->id – square’s ID

- ->x, ->y – square coordinates on the map
- ->town – square's town ID
- ->getZombies($aDay) – returns an Application_Model_Zombies object for a given day or an array of objects if $aDay is null.
- ->hasZombies($aDay) – returns TRUE if there is information on zombies for day $aDay, FALSE otherwise.

- Application_Model_Zombies – the class represents zombies associated with a zone. It has the following properties and methods:
  - ->id – object's ID
  - ->squareId – ID of the square it's associated with
  - ->day – game day the object is associated to
  - ->zombiesMin – minimum amount of zombies in the square or null if the information is accurate
  - ->zombiesMax – maximum amount of zombies in the square
  - ->baseZombiesMin – minimum base zombie count (base zombies are the zombies originally found in the zone, before any killing) in the square or null if the information is accurate
  - ->baseZombiesMax – maximum base zombie count (base zombies are the zombies originally found in the zone, before any killing) in the square
  - -> getZombies() – returns zombiesMax if information is accurate or array("min"=>zombiesMin, "max"=>zombiesMax) if it isn't accurate
  - ->getBaseZombies() - returns baseZombiesMax if information is accurate or array("min"=>baseZombiesMin, "max"=>baseZombiesMax) if it isn't accurate
  - ->isAccurate() – returns TRUE if zombie information is accurate, FALSE otherwise
  - ->isBaseAccurate() – returns TRUE if base zombie information is accurate, FALSE otherwise
  - ->difference() – returns the difference between baseZombiesMax and zombiesMax, regardless of information accuracy. This is the amount of zombies killed.

2.6.Example plug-in code

```
<?php

require_once 'ModuleInterface.php';

class Module_Predict_PredictModule implements ModuleInterface
{
	private $mAtlas;
//	initialize the module
	public function Initialize() {
		$this->mAtlas = Application_Model_Atlas::getInstance();
	}
//	return prediction data
	public function GetSquareData(Application_Model_Square $aSquare) {
		$lZombies = $aSquare->getZombies($this->mAtlas->day);
		$lRes = array();

		if (! is_null($lPredictions = $this->getAllPredictions($aSquare))) {
			foreach ($lPredictions as $lPrediction) {
				$lRes["predictions"]["day".$lPrediction->day] = $lPrediction->getZombies();
```

```
                  }
            }
            return $lRes;
      }
public function SquareCreated (Application_Model_Square $aSquare,
SimpleXMLElement $aXML) {
      $lNewPrediction = new Module_Predict_Predictions();
      $lNewPrediction->setSquareId($aSquare->id)
                        ->setDay($this->mAtlas->day);
      $lZombieMin = round($lNewPrediction->day * 0.8);
      $lZombieMax = round($lNewPrediction->day * 1.2);
      if ($lZombieMin == $lZombieMax) $lZombieMin = null;
      $lNewPrediction->setZombiesMin($lZombieMin);
      $lNewPrediction->setZombiesMax($lZombieMax);
      $this->savePrediction($lNewPrediction);
}

public function HandleForwardedRequest() {
      return array("PredictData"=>"Success!");
}
      //    application tabs
      public function HasOwnTab () {return FALSE;}
      public function GetTabName () {}
      public function GetTabContent () {}

      //    main tab content
      public function PreMapHTML () {}
      public function PostMapHTML () {
            if
(file_exists(APPLICATION_PATH."/modules/Predict/post.phtml"))
                  return
file_get_contents(APPLICATION_PATH."/modules/Predict/post.phtml");
      }

//    square update form element added
      public function GetFormElements () {
            $lElement       =       new       Zend_Form_Element_Text("kills",
array("label"=>"Kills"));
            return array($lElement);
      }
//    update predictions based on used data, return updates square
coordinates plus All of the squares around it (their predictions may have
changed)
      public function UpdateFormSubmitted (Zend_Form $aForm) {
            $lChanges = array();
        $lValues = $aForm->getValues();
            $lX = (int) $lValues["x"];
      $lY = (int) $lValues["y"];
      $this->updatePredictions( $this->mAtlas->getSquare($lX, $lY) );
      if (!is_null($this->mAtlas->getSquare($lX-1, $lY))) {
             $this->updatePredictions( $this->mAtlas->getSquare($lX-1, $lY)
);
            $lChanges[] = array("x"=>$lX-1, "y"=>$lY);
      }
      if (!is_null($this->mAtlas->getSquare($lX+1, $lY))) {
             $this->updatePredictions( $this->mAtlas->getSquare($lX+1, $lY)
);
            $lChanges[] = array("x"=>$lX+1, "y"=>$lY);
      }
      if (!is_null($this->mAtlas->getSquare($lX, $lY-1))) {
```

```
              $this->updatePredictions( $this->mAtlas->getSquare($lX, $lY-1)
);
            $lChanges[] = array("x"=>$lX, "y"=>$lY-1);
      }
      if (!is_null($this->mAtlas->getSquare($lX, $lY+1))) {
              $this->updatePredictions( $this->mAtlas->getSquare($lX, $lY+1)
);
            $lChanges[] = array("x"=>$lX, "y"=>$lY+1);
      }
      return $lChanges;
      }
//    XML was updated, update all predictions
      public function XMLUpdated (SimpleXMLElement $aXML) {
        $lTownData = $this->mAtlas->XML->data->city;
            for  ($x=(0  -  $lTownData["x"]);  $x<($this->mAtlas->mapSize  -
$lTownData["x"]); $x++) {
                  for  ($y=($lTownData["y"]  -  $this->mAtlas->mapSize  +  1);
$y<($lTownData["y"] + 1); $y++) {
                  $lSquare = $this->mAtlas->getSquare($x, $y);
                  $this->updatePredictions($lSquare);
                }
            }
      }
}

… (private methods)

?>
```

2.7.

3. Browser-side Atlas API

The browser –side plug-in development is made using jQuery framework. The work is wrapped around a global Atlas object providing an interface to all data associated with the platform, game and map, as well as some hooks to common actions performed by users.

All function defined within the platform should be members of the Atlas global `var` to prevent namespace pollution. After a call to registerModule(name) a subobject is created (`Atlas.modules.name = {}`). All function should be defined as members on that object (e.g. `Atlas.modules.Predict.someFunction = function() {…}`).

3.1.Attributes and methods

Below is a list of all common attributes and methods, that can be called directly in any aread of the code. Special methods related to special events that can be hooked into are described later (see 3.4 Hooks for details).

- Attributes
  - XMLFeed – an object with XML feed properties. So far this includes:
    - game – information from the <game> tag (id, days, datetime, quarantine, etc.)
    - city – information from the <city> tag (city, door, water, etc.)
    - The above information can be accesses through standard object properties (e.g. Atlas.XML.city.water)
  - squares – an array of square objects, indexed with x, y coordinates (e.g. var lSquare = Atlas.squares[x][y];)
  - day – current game day
  - mapSize – map row/column count
  - townCoordinates – x, y coordinates of the town square, from the XML feed, as an object with .x and .y properties. Useful when converting feed data to relative coordinates.
- Methods
  - registerModule(name) – registers module name with Atlas.
  - appendForUpdate(coordinates) – appends a square for update, takes an object with .x and .y properties (e.g. {x:1,y:3}). This requires a call to UpdateNext() global function to start the update process. Useful when more than one zone is added for update in a row.
  - updateZone(coordinates) – updates a square, takes an object with .x and .y properties (e.g. {x:1,y:3}). Calls UpdateNext() automatically. Use when only one square should be updated.
  - dangerLevel(n) – returns a danger level for n zombies as an integer <0-4>
  - waiting(message) – displays the waiting icon with message alongside
  - waitingDone() – should be called after an operation that started the waiting() is done to hide the icon

3.2.Helper functions

Below is a list of helper functions available in the Atlas object.

- Atlas.helpers.UpdateNext() – starts the square update queue (see: Atlas.appendForUpdate() and Atlas.updateZone())
- Atlas.helpers.alertProps (obj) – creates an alert box with all object properties. Used only for debugging!

3.3.Square object

A square object has properties acquired from the PHP side of a plug-in and some core properties that shouldn't be altered. After each call to the Atlas.SquareDataRecieved() (see 3.4 Hooks for details) all data received is assigned as object properties and is later available as such. Below is a list of standard properties available for all squares:

- .x, .y – square coordinates, relative to town position (f.e. ranging from -10 to 14, not 0 to 25 like in the XML)
- .HTML – the HTML representation of the square (a <td> element at this point) as a jQuery object (all jQuery methods are available for it). DO NOT OVERWRITE!
- .base_zombies, .zombies – zombie information, as int if it's accurate, as an object (.min and .max properties) if it's not accurate. Use `if (typeof(square.zombies) == "object")` to differentiate between the two.

All additional properties returned from the server (see: 2.3 ModuleInterface - GetSquareData() and 3.4 Hooks for details) will be added as additional square object properties.

3.4.Hooks

The below methods are special and are called on certain events within the application. All of them have an `append` method, which should be used like this:

```
Atlas.appendOnLoad( function () {
      Atlas.modules.Predict.PredictPrepareMap();
      [...]
      }
} );
```

They all take a function as a parameter, some with parameters themselves. Methods receiving an event as a parameter can do the following to retrieve square related to the event:

```
var zone = event.data.zone;
var lSquare = Atlas.squares[$(zone).data("x")][$(zone).data("y")];
```

The hook methods should not be called directly.

- onLoad() – executed when the DOM tree is loaded and ready to be manipulated. All initialization code should go here.
- onSquareClick(event) – executed when a map square is clicked. By default it shows the update form and fills it with data.
- onSquareMouseOver(event) – executed when mouse pointer enters a map square. By default it fills the popup window with square information.
- onSquareMouseOut(event) – executed when mouse pointer leaves a map square. By default it hides the popup.

- onSquareMouseMove(event) – executed when mouse moves over a square. Bu default it moves the popup window.
- XMLUpdate (xml) – executed when XML contents is updated, either explicitly by calling update-feed AJAX or when new XML is received from get-changes AJAX call.
- SquareDataRecieved(aData) – executed when square data is received (both on page initialization and all consecutive ajax updates). aData contains all information returned by PHP GetSquareData() (see: 2.3 - GetSquareData() for details) as an object. By default all properties are assigned to a square object:

```
var lSquare = Atlas.squares[aData.x][aData.y];
for (var key in aData) {
      lSquare[key] = aData[key];
}
```

3.5.Example plug-in code

```
Atlas.registerModule(„Predict”);

Atlas.appendOnLoad( function () {
      Atlas.modules.Predict.PredictPrepareMap();
      var lZones = $(".PredictZone");
      for (var i=0;i<lZones.length;i++) {
            $(lZones[i]).mouseover ({zone: lZones[i]}, function(event) {
Atlas.modules.Predict.FillPredictionPopup(event);});
            $(lZones[i]).mousemove ({zone: lZones[i]}, function(event)
{Atlas.onSquareMouseMove (event);});
            $(lZones[i]).mouseout ({zone: lZones[i]}, function(event)
{$('#zonepopup').hide();});
      }
} );

Atlas.appendSquareDataRecieved( function(aData) {
      var lSquare = Atlas.squares[aData.x][aData.y];
      if (lSquare.predictions["day"+Atlas.day] != undefined) {
            var lPrediction = lSquare.predictions["day"+Atlas.day];
            if (lSquare.HTML.hasClass("undefined")) {
                  lSquare.HTML.removeClass("undefined");
                  if (typeof(lPrediction) == "object") {
                        lSquare.HTML.text(lPrediction.min+"                -
"+lPrediction.max+"*");

      lSquare.HTML.addClass("danger"+Atlas.dangerLevel(lPrediction.max));
                  }
                  else {
                        lSquare.HTML.text(lPrediction+"*");

      lSquare.HTML.addClass("danger"+Atlas.dangerLevel(lPrediction));
                  }
            }
      }
      Atlas.modules.Predict.UpdatePredictionSquare(aData.x, aData.y);
});

Atlas.appendOnSquareMouseOver( function (event) {
      var zone = event.data.zone;
      var lSquare = Atlas.squares[$(zone).data("x")][$(zone).data("y")];
      if (lSquare.predictions["day"+Atlas.day] != undefined) {
            var lPrediction = lSquare.predictions["day"+Atlas.day];
```

```
            var lPp = $("<p>");
            if (typeof(lPrediction) == "object") {lPp.text ("Zombies
(predicted!):"+lPrediction.min+" - "+lPrediction.max);}
            else {lPp.text ("Zombies (predicted!):"+lPrediction);}
            $('#zonepopup #content').append(lPp);
      }
} );

Atlas.modules.Predict.PredictPrepareMap = function () {
      var lContainer = $("#PredictionsContainer");

      var lTitle = $("<div>");
      lTitle.text("Prediction for tomorrow");
      [...]
}

Atlas.modules.Predict.UpdatePredictionSquare = function (x, y) {
      var lSquare = Atlas.squares[x][y];
      if (lSquare.PredictHTML == undefined)
            return;
      var lDay = parseInt(Atlas.day) + 1;
      lSquare.PredictHTML.removeClass("unknown danger0 danger1 danger2
danger3 danger4");
      if (lSquare.PredictHTML.hasClass("town")) {

      }
      else if (lSquare.predictions["day"+lDay] != undefined) {
            lPrediction = lSquare.predictions["day"+lDay];
            if (typeof(lPrediction) == "object") {
                  lSquare.PredictHTML.text(lPrediction.min+"-
"+lPrediction.max);


lSquare.PredictHTML.addClass("danger"+Atlas.dangerLevel(lPrediction.max));
            }
            else {
                  lSquare.PredictHTML.text(lPrediction);


lSquare.PredictHTML.addClass("danger"+Atlas.dangerLevel(lPrediction));
            }
      }
      else {
            lSquare.PredictHTML.text("?");
            lSquare.PredictHTML.addClass("undefined");
      }
}

Atlas.modules.Predict.FillPredictionPopup = function (event) {
      var zone = event.data.zone;
      var lSquare = Atlas.squares[$(zone).data("x")][$(zone).data("y")];
      $('#zonepopup                                #header').text("Coodrinates:
"+lSquare.x+","+lSquare.y);
      $('#zonepopup #content').text("");
      var lDay = parseInt(Atlas.day) + 1;
      if (lSquare.predictions["day"+lDay] != undefined) {
            var lPrediction = lSquare.predictions["day"+lDay];
            var lPp = $("<p>");
            if (typeof(lPrediction) == "object") {lPp.text ("Predicted
zombie count:"+lPrediction.min+" - "+lPrediction.max);}
            else {lPp.text ("Predicted zombie count:"+lPrediction);}
```

```
            $('#zonepopup #content').append(lPp);
        }
        $('#zonepopup').show();
}
```