University of Zurich^UZH

SpaghettiLens

—

Lens modeling made easy

Technical Report

by Rafael Kueng

September 2013

rafik@physik.uzh.ch

Contents

1 Introduction

Gravitational lenses (GLs) are a great tool for astronomers to study properties of the cosmos. But before they can be used for further studies, they have to be spotted.

This is a hard task, given the huge amount of data which observatories produce and the average performance of existing robotic lens detection tools. SpaceWarps[1] addresses this problem by inviting volunteers to help spot GLs, with a great success. Over 1.7 million images have been classified in the first two weeks after the start of the citizen science project.

The next step is to analyze and model the lens candidates found. A convincing model for a lens candidate supports the hypothesis that it is really a lens. Further, it is the first step in a more detailed analysis of the lens. It allows for example the analysis of the matter distribution including dark matter.

Creating a model is a rather sophisticated, time intensive task usually done by scientists. We proposed that it is possible for non scientists (volunteers[2]) to create models that are comparable to those of scientist. This requires in a first step to present the underlying theory in a compact and simple way. Second, a modeling tool has to be developed that simplifies the process of creating a model. It would be preferable, if this tool provides an instant, meaningful visual output. That would allow the volunteers to improve their models iteratively, by try and error. In a third step, the building of a community of users should be encouraged. On one side by closely working together to further adjust the tool to the users needs, on the other side to assist if any questions come up.

This report introduces the tool written for this task: SpaghettiLens. It was part of the authors Master Thesis and details the design and set up of the tool. This is intended to be a technical documentation for anybody who wants to understand and contribute to the application.

The scientific part of my Master Thesis is to test our proposal by letting volunteers model simulated lenses. The known parameters of the simulations were compared to the parameters of the models generated by the volunteers. The results of this part can be found in the scientific paper (to be published, you can have a look at the draft at `http://www.physik.uzh.ch/~rafik/?f=slp.pdf`)

Additionally, a tutorial video was recorded, which explains the theory and use of SpaghettiLens[3].

1.1 SpaghettiLens

SpaghettiLens is build around GLASS[5], a non parametric, pixel based lens modeling tool set. GLASS is a reimplemented version of PixeLens[13] SpaghettiLens is build as a client-server web application that runs on any computer in the browser without any

[1]`http://www.spacewarps.org`
[2]also known as citizen scientists
[3]`http://mite.physik.uzh.ch/tutorial/`

installation[4]. If offers the following features:

- Quickly create models of lenses, originating from different data sources (Space-Warps, MasterLens[5])
- Users can easily present their results, discuss and improve them
- Allows scientists to store, manage and evaluate models and data for lenses

It was designed using modern web technologies to implement a powerful, modular web application. Chapter 2 gives an introduction to current web technologies applied in this project. The main part of this report is Chapter 3, explaining all the parts SpaghettiLens is made of in detail, whereas Chapter 4 details the interplay between the modules. Chapter 5 gives a short overview over the installation and maintenance and finally some review and outlook in Chapter 6.

[4]Besides an up to date browser

[5]`http://www.masterlens.org`

2 Primer in Web Technologies

This section gives a short overview over the techniques and standards used in web development. It introduces all the concept needed to develop an dynamic web application like SpaghettiLens.

2.1 HyperText Markup Language (HTML)

The HyperText Markup Language (HTML) is the standardized markup language. It is used as a format to exchange semantically structured information on the internet using the HyperText transfer protocol (HTTP). It uses plain text to describe the building blocks of a web site in a tree like structure. Those building blocks are called `tag` and can have a set of attributes, expressed as key–value pairs. They are enclosed in angle brackets and usually consist of opening tag, the attribute set (for example `id`, `class`) and an closing tag. Tags can be nested or contain plain text data to be displayed. See Listing 2.1 for an example of a HTML document.

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Title of the document</title>
  </head>
  <body>
    <p id="id1" class="class1 class2">Hello world!</p>
    <span id="id2" class="class1 class2">...</span>
    <div id="id3" class="class1">Good Bye!</div>
  </body>
</html>
```

Listing 2.1: Example of a HTML5 document.

2.1.1 The Document Object Model (DOM)

The nested design of HTML results in a tree like structure, the document tree. It can be accessed using the interface definition Document Object Model (DOM). Often, the document tree itself is referred to as DOM.

2.1.2 HTML4 vs HTML5

There are a few versions and variants of standardized HTML available. The actual standard is ISO/IEC 15445:2000[11], based on HTML 4.01, published by W3C in May, 2001.

The new version, HTML5 is not yet finalized. The first working draft was published 2008[9]. It offers many new features, including the Canvas element (discussed in Section 2.5.2) and the integration of SVG (Section 2.5.3).

2.2 Cascading Style Sheets (CSS)

Cascading Style Sheets (CSS) is a language to describe the layout of structured semantic documents, like HTML. It allows the identification of elements of a document by tag name, id, class name, or (relative) position in the document by applying powerful CSS selector syntax. Selected elements are key – value pairs applied to that define the optics and layout of the selection.

```css
span { /* select by tag */
  font-weight: bold;
}
#id3 { /* select by id */
  border: 1px solid black;
}
.class1 { /* select by class */
  display: block;
}
p.class2{ /* select element with 'p' tag AND class 'class2' */
  font-color: green;
}
```

Listing 2.2: Example CSS file. Operating on Listing 2.1

The current version is CSS 2.1[3], defined in June 2011. The work on CSS 3 is on going and rather advanced[2]. Many features of CSS 3 can already be used[6].

2.3 ECMAScript (aka JavaScript JS)

JavaScript (JS) is an interpreted, prototype based multi-paradigm scripting language. It usually runs on the client side in the browser. It's purpose it to alter the DOM, to react to the user and to interact with a server.

JS has no concept of classes, even though it offers object oriented programming by using prototype based programming. JS is dynamically typed and allows duck typing[6]. JS objects are a collection of key – value pairs and thus comparable to Python `dict` and C++ `map`.

The core components of JavaScript are standardized as ECMAScript (ECMA 262 [7]), ISO/IEC 16262 [10].

[6]Variables have no fixed data type assigned

```
var id1 = document.getElementById('id1');

id1.onClick = function(evt){
  var id3 = document.getElementsByTagName('div')[0];
  id3.className += ' class2';
  id3.innerHTML = 'Don\'t leave yet!';
}
```

Listing 2.3: JS code modifying the above HTML file Listing 2.1. Makes the first element clickable. If the user clicks on the first element, the third elements text is changed to “Don’t leave yet!” and made green.

2.3.1 jQuery library

jQuery is an open source library that simplifies the access to and manipulation of DOM elements. It allows the use of CSS selectors to easily select and modify a extended set of DOM elements, an extended event system, animations and effects and Ajax functions, among others. It simplifies the writing of JavaScript code by offering shorter syntax and browser-independent high level functions. Compare Listing 2.3 to Listing 2.4. jQuery is used by about two third of the top 10’000 internet sites[4].

```
$("#id1").onClick(function(evt){
  $("div").first()
    .addClass("class2")
    .text("Don't leave yet!");
};)
```

Listing 2.4: The same functionality as in Listing 2.3, using jQuery JavaScript library.

2.3.2 Security

JavaScript code can potentially be harmful to a user. To prevent exploits, JS is usually run encapsulated in a sand box by the browser. This sand box prevents access to all local resources, including the file system[7], and other open web pages. This implies that users manipulating data in an web app can’t save states on their local machine easily. Web browsers further impose a strict security restriction on content loaded by JS from sites that don’t have the same origin as the currently displayed web site (“same-origin policy”).

[7]Except for cookies, where a few bytes can be saved

2.4 Dynamic web based applications (web apps)

2.4.1 Dynamic HTML (DHTML)

A regular, static HTML web page gets loaded by the browser from the server in order to display data to the user and offer links that lead to other web sites, triggering a new load-render-display cycle. The umbrella term DHTML describes web design techniques that allow a web page to modify it's structure based on user input, without a reload. This requires JS to modify the DOM, if an user event happens. A change in the DOM triggers a refresh of the page, but not a reload.

This techniques allow a web page to become a dynamic application, which maintains an internal state that can be changed by user applications.

2.4.2 Server Side Scripts

Using HTML a web server can only serve static data to any client. Usually, there is a need of a client to get dynamically data from a server side data source, like a data base, and to put manipulated data back in. Server side scripts provide an interface to those data sources to client applications. When the user makes a request for a resource like a HTML file, this scripts run on the server, produce the desired data on the fly by accessing the data sources needed and send back the rendered data.

A server side script can be written in any programming or scripting language that compiles and runs on the server operating system and can interact with the web server software. There are many server side scripting languages available, proprietary and open source. Some of the most popular ones are Active Server Pages (ASP, proprietary), Java Server Pages (JSP), Perl, PHP: Hypertext Preprocessor (PHP), Python (using Framework Django) and Ruby on Rails (all open source).

Listing 2.5 shows a HTML document with embedded server side script code in PHP. A client calling this page will trigger the server side rendering of this page, resulting in Listing 2.6 that is then being sent to the client.

2.4.3 Client to Server communication

Basic data transfer on the internet is handled using the HyperText Transfer Protocol (HTTP). A client (browser) always initiates a TCP connection, requesting a resource identified by an URI[8]. The server sends an answer using the same connection and closes it afterwords. If the request was successful, the client renders and displays the body of the answer, the HTML document. Not only HTML documents are transmitted. The same technique is used to transmit CSS, JS, images or any other type of file, text or data.

There are different types of requests possible, of which the most important ones are GET and POST. Basic GET requests are used to request resources from a server. They can also be used to send data to the server, attaching key – value pairs to the URI[9].

[8]Uniform Resource Identifier

[9]`www.url.net/index.html?key1=value1&key2=value2`

```php
<?php
  function getDayOfWeek() {
    return date('l');
  }
?>

<!DOCTYPE html>
<html>
  <head>
    <title>Title of the document</title>
  </head>
  <body>
    <?php
      echo "<p>Today is: " . getDayOfWeek() . "</p>\n";
    ?>
  </body>
</html>
```

Listing 2.5: Example server side script in PHP

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Title of the document</title>
  </head>
  <body>
    <p>Today is: Sunday</p>
  </body>
</html>
```

Listing 2.6: Result of server side script 2.5 if rendered on a Sunday.

The amount of data that can be transferred with GET is restricted to about 255 bytes. A `POST` request message consists, in opposite to a GET request, in a header and a body. The message body contains the data to be sent and is therefore of unrestricted size.

In order to be able to combine server side scripts and DHTML, the web browser has to expose an API that allows JS to send HTTP requests and get the response, without triggering an automatic rendering and update of the screen. This API is called XMLHttpRequest (XHR). Despite the name, any text based data format can be sent and received with XHRs: JSON, HTML, XML, or plain text.

JavaScript Object Notation (JSON) is a simple plain text data format used for easy exchange of data. It represents a JS objects key – value pair, written down using JS syntax. Refer to Listing 2.7 for an example.

```
{
  "user":"John Doe",
  "password":"sEcReT",
  "age":42,
  "pages":[3,1,4],
}
```

Listing 2.7: A simple JSON object.

2.4.4 Asynchronous Javascript And XML (AJAX)

The combination of all the described techniques in Section 2.4 results in the concept of Asynchronous Javascript And XML (AJAX). AJAX applications feature a client side JS engine that decouples user interaction with the client side application (DOM manipulation) and the communication of the client and the server, which happens in the background. The resulting web app thus feels as fluent as a native desktop application.

The JS engine does most of the data processing on the client side, while asynchronously sending and loading data to the server in the background using XHR.

2.4.5 Security and Restrictions

Since web servers can not establish a connection to a client, the only way for a client to get asynchronous events from the server is by repeatedly sending requests to check for new server side events. This technique is called "polling". There are workarounds to implement a push functionality. One being long polling, where the client sends a request that the server does not answer instantaneously. Therefore the connection is kept open, in order to send data when available. Such setups are possible, but they are not standard compliant and often involve some modification / hacks of web servers. The coming HTML5 standard proposes mechanisms which implement push features, like Server Sent Events and the WebSocket API. These mechanisms are still experimental and not widely implemented.

A major restriction is the "same-origin security policy". The policy prohibits JS and XHR access to content that is not hosted on the same server. For example, a client application can not use XHR to load images from a web site other than the applications origin. There are workarounds for this issue. Set up a server side script that acts as proxy for an access to external resources. This increases the load on the server. Using the standardized mechanism of Cross-Origin Resource Sharing (CORS), a web server admin can allow foreign XHR requests to his page.

2.5 Interactive Graphics

To produce a webapp, often a custom user interface and some sort of dynamic, interactive graphics system is needed. HTML in combination with CSS and static images offer only very limited abilities to design custom dynamic screen elements. Scalable Vector graphics (SVG) and HTML5 canvas offer more powerful techniques to create dynamic screen content.

2.5.1 HTML images

All HTML DOM elements are rectangles and can be placed anywhere on the screen using CSS. This includes the image tag `<img>` that allows a set of images to be composed to an interface. JS can be used to change the position of DOM elements, or to change the image file they display. But HTML4 offers no abilities to change the contents of a particular image. This means you have to create all the images that possibly could be displayed. This is fine for simple situations like a menus, but is not practical for an advanced user interface.

2.5.2 HTML5 canvas

The HTML5 specification defines a new tag, called "Canvas". It allows pixel based image manipulation, by offering a canvas that consists of a discrete, rasterized 2D array of pixels. By using JS, each pixel of this image grid can be accessed and it's color can be changed dynamically. There are supportive functions that allow the drawing of lines, shapes, paths and images.

Canvas offers fast, low level pixel wise image manipulation. It is ideal for image manipulation on a per pixel basis like blending.

It offers no scene graph that keeps track of which structures and shapes were used to draw on the canvas. If one element changes its position, the canvas needs to be deleted and all objects need to be painted again. The canvas offers JS callbacks for user actions like click, returning the coordinates of the pixel that was clicked on. The programmer has to implement a scene graph himself if he wants to know what object / element was clicked on.

A rendered canvas can be converted to a pixel image using the implemented function `.toDataURL()`, given the Canvas was not tainted. A Canvas gets tainted if any external resource gets drawn in the Canvas, offending the same origin policy described in Section 2.4.5.

2.5.3 Scalable Vector Graphics (SVG)

Scalable Vector Graphics (SVG) is a vector image format standard, using a markup language (similar to HTML, based on XML). SVG defines a continuous coordinate system, on which mathematical constructs are defined. These mathematical constructs are objects like paths, basic shapes, test, raster graphics that are directly stored as a SVG tag in the SVG file. Since SVG is an XML language, it is comparable with HTML. CSS can be used to define styles for the SVG objects and JS can be used to manipulate properties.

SVG files offer high level access to graphics, based on shapes. The direct access to shapes with JS makes it easy to dynamically modify elements or add new shapes. The SVG DOM can be directly included in the HTML DOM, offering unified access from JS to all HTML and SVG objects. JS events get fired like in regular HTML. For example each shape fires an `onClick` event if clicked on.

By design, SVG is slower as canvas, because it first needs to be rasterized by the browser in order to be drawn on the screen. Additionally, there was put a lot of effort in Canvas optimization and hardware acceleration lately, whereas SVG hardware support is only minimal. Due to the lack of hardware optimization, pixel based operations like filters operate slowly.

SVG graphics can be rendered to a canvas using the external JS library `canvg.js`[10].

[10]Developped by Gabe Lerner with contributions to the CSS engine by Rafael Küng and others.

3 Implementation of SpaghettiLens

SpaghettiLens applies the concept of an AJAX web application, as described in Section 2.4.4. It consists of several modules which communicate through defined APIs. Figure 1 shows an overview of the implemented modules and their communication channels.

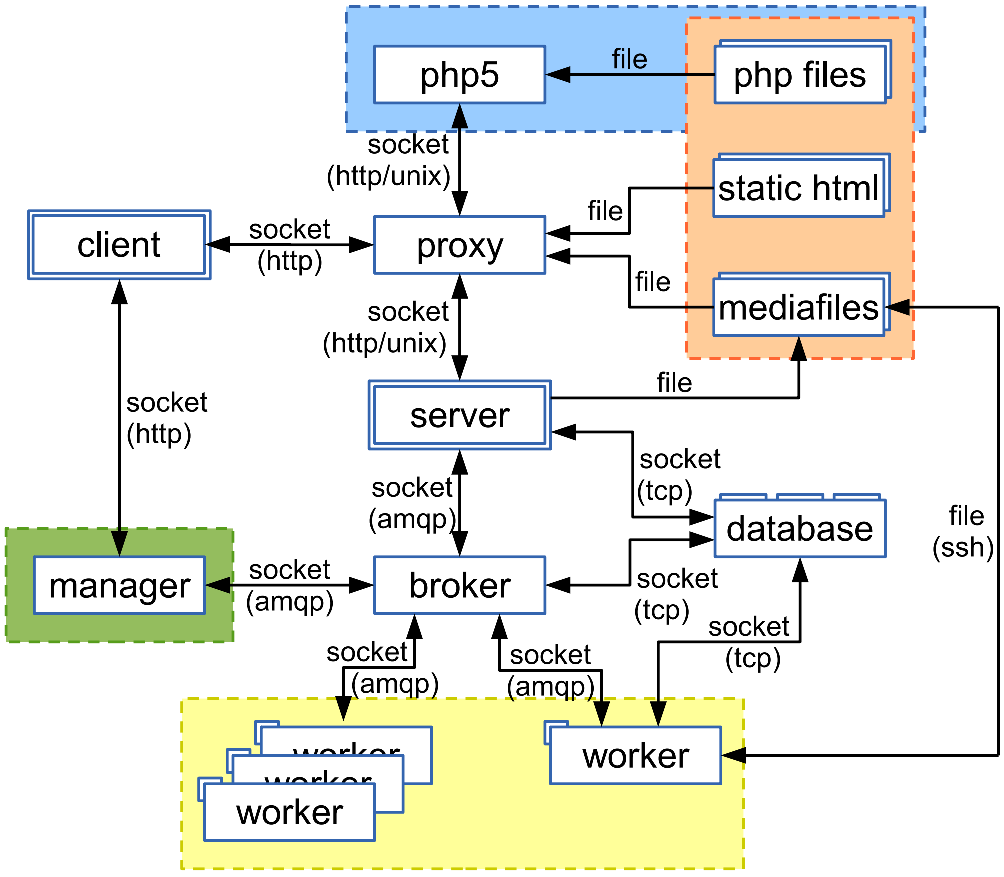


Figure 1: Schematics of full SpaghettiLens setup including communication protocol. Including additional module for hosting additional pages (top blue box) and administrative interface for worker management (left green box). Also showing the file system (top right orange box) and worker nodes running the simulation software (bottom yellow box).

Figure 1 shows all modules used in the current setup as an web application at `http://mite.physik.uzh.ch/`. The client, server and worker modules are the core components, needed to run SpaghettiLens. Simpler setups are possible. SpaghettiLens could be run as a native desktop application using only the core modules server and client.

One advantage of this modularized approach is the scalability of the application. Each module can possibly run on it's own hardware, if the use of the application increases.

The application features a simple flow of data. After the loading of a Lens into the client application on the browser (client), the user creates a model, which is represented as a JSON string. This model is sent to the server side application, where the model and all according parameters are saved in a data base. Additionally, the server creates a configuration file (`.cfg` file) which is stored in the file system so that the simulation application (worker) con run on it. The worker runs the simulation and the resulting set of models is saved to the file system as a State file (`.state`). A second worker step (renderer) then loads this state file and creates the resulting images (mediafiles) and saves them to the file system. In the current implementation, the simulation and the rendering are done by the same worker. Future development will feature a separate renderer, in order to allow the creation of arbitrary plots from already simulated data. In a last step, the client loads the rendered images and gives the user a visual feedback of the model created.

Figure 2 shows this high level view of data flow. The sections in this chapter elaborate the interfaces for this data flow, the design of according data structures, as well as the mechanics and implementation of the modules. Section 4 shows the program and data flows in more detail.

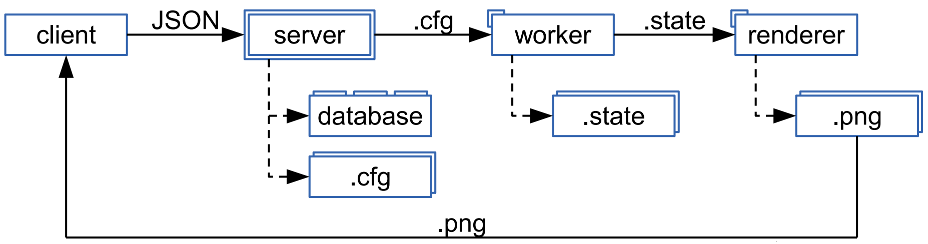


Figure 2: High level view of flow of data involved in creating a model and data objects saved.

3.1 Client-side Application (Client)

The client-side application (client) was programmed using a modular, strict object oriented, event driven approach. The four main parts of the client application are the Engine Core (engine), the User Interface (UI), the Application State (state) and the Communication Interface (com). Figure 3 shows the clients modules and according JS object namespaces.

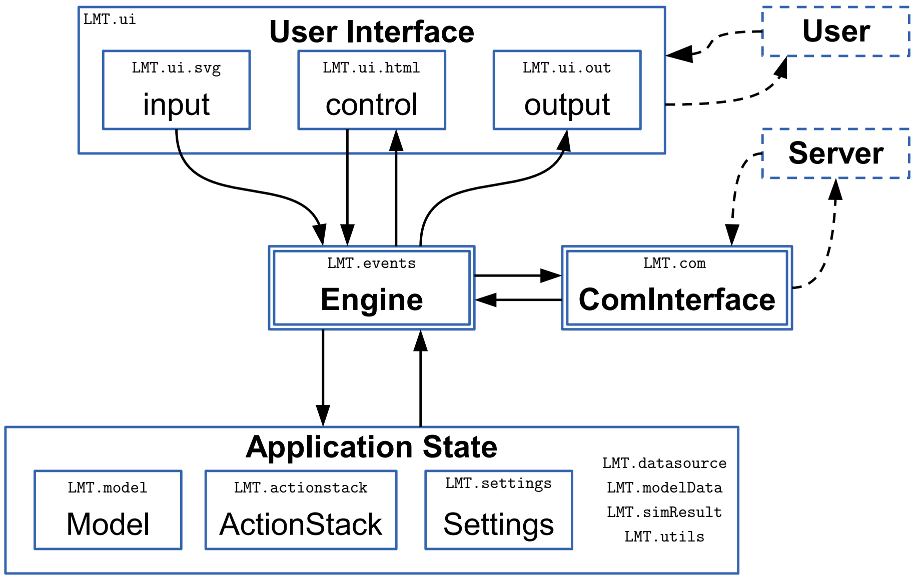


Figure 3: Modules of the client-side application and according JS namespaces

The object oriented paradigm states that all data and actions that affect similar structures should be grouped together to a functional unit, named object. Such an object has internal data representing the objects state. Further, it offers functions called "methods" that allow other objects to get or to change the internal state of the object.

Event driven paradigm implies that the program flow is established using messages, called events. Those can be fired by any object, called the "event source" (the "UI"[11] or the "Com"[12]). Event handlers (also known as "listener", "receiver" or "observer") are objects that react to a certain event. The main loop of the program, the event engine, takes care of detecting events and calling assigned handlers. Any object can be event source and handler at the same time.

3.1.1 Engine

Since JS running in a browser is by default event driven, the main loop and some rudimentary features are implemented by the browser. jQuery enhances this capabilities and allows easy registration of event handlers and firing of events.

[11]click on button
[12]received data

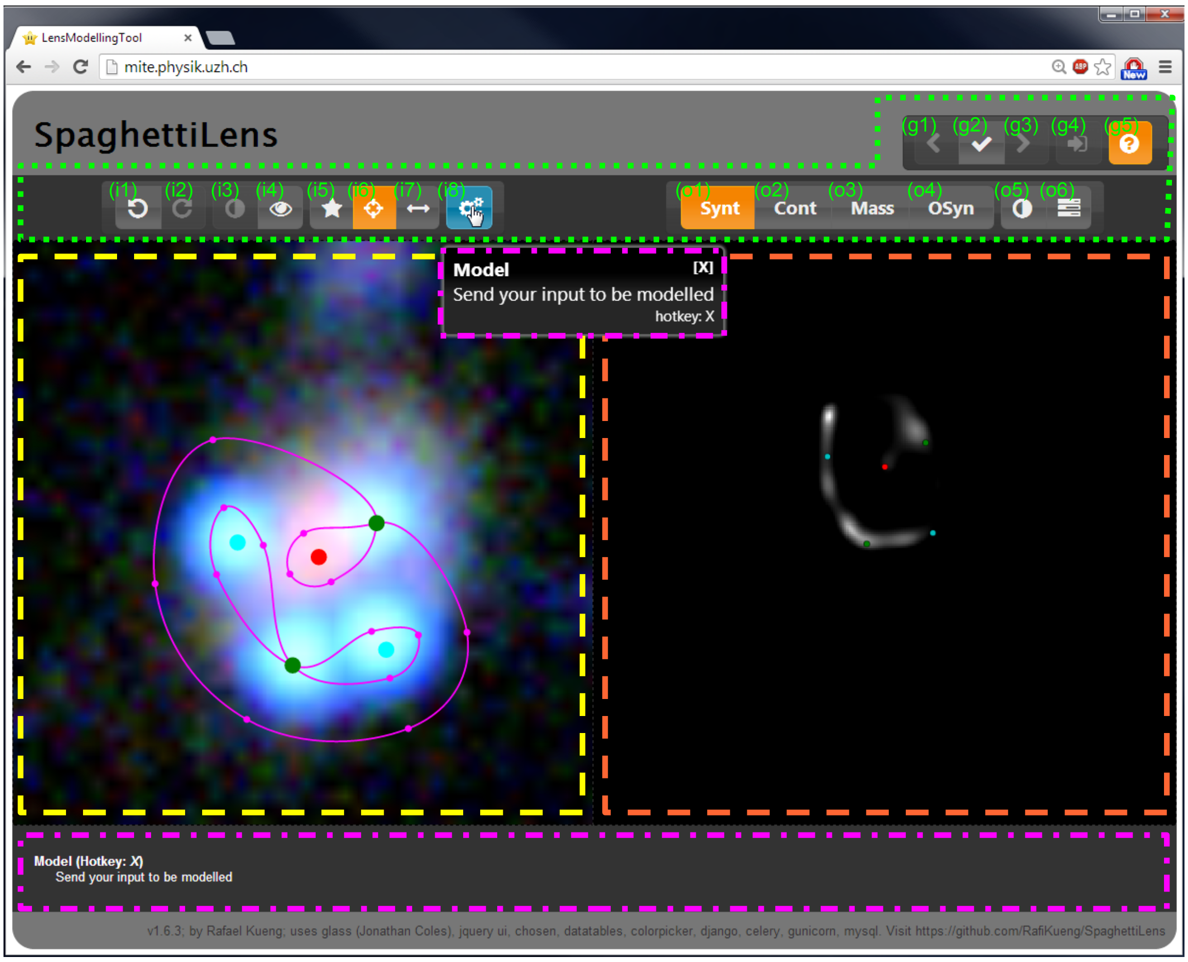
The engine of SpaghettiLens, the object `LMT.events`, defines all possible events on which the application reacts by registering all event handlers at start up. It further initializes all data objects defining the state of the program, see Section 3.1.4. Listing 3.1 shows an excerpt.

```
LMT.events = {
  startUp: function(){
    //...
    LMT.model = new LMT.objects.Model();
    LMT.model.init();
    LMT.events.assignHandlers();
    //...
  },

  assignHandlers: function() {
    // ...
    $(document).on('ShowSelectDatasourceDialog',
      LMT.ui.html.SelectDatasourceDialog.show);

    $(document).on('RepaintModel', LMT.objects.Model.Repaint);

    $(document).on('UploadModel', LMT.com.UploadModel);
    $(document).on('SimulateModel', LMT.events.SimulateModel);

    $(document).on('ReceivedSimulation', LMT.ui.out.load);
    $(document).on('ReceivedSimulation', LMT.ui.html.Toolbar.updateTop);
    // ...
  },
}
//...
```

Listing 3.1: The Engine `LMT.events`. An excerpt from *`lmt.events.js`*.

3.1.2 User Interface (UI) – Visual Design

The basic design idea for the User Interface (UI) was to provide a visual feedback for the modeling process. Figure 4 shows a screenshot of the UI. It allows for easy comparison between the model parameters in the input area (on the left hand side, yellow dashed line) and several resulting images generated from the simulated model on the output screen (right hand side, orange dashed line). All tools needed to create and manipulate the input are arranged on top of it. Tools for manipulating the simulation process and the output view are placed above the right hand side. A few general commands are arranged in a top bar (all marked with green dotted line).



Model [X]
Send your input to be modelled
hotkey: X

Model (Hotkey: X)
Send your input to be modelled

v1.6.3; by Rafael Kueng; uses glass (Jonathan Coles), jquery ui, chosen, datatables, colorpicker, django, celery, gunicorn, mysql. Visit https://github.com/RafiKueng/SpaghettiLens

Figure 4: User Interface. Depicting input area (yellow dashed left), output area (orange dashed right), general input (toolbar, green dotted) and two help areas (magenta dot dashed, top mouse-over help, bottom static help window). Toolbar buttons: (g1) Load previous lens in catalog (g2) Save and upload the model (g3) Load next lens in catalog (g4) Log in / create user account (g5) Toggle static help bar (i1) Undo (i2) Redo (i3) Change brightness / contrast of input background image (i4) Change display settings (i5) Point mass identification mode (i6) Lensed image identification mode (i7) Ruler tool (i8) Simulate current model (o1) Show a synthetic image (interpolated) (o2) Show contour lines (o3) Show the mass distribution (o4) Show the original synthetic image (o5) Change brightness / contrast of output background image (o6) Change advanced simulation parameters

To assist the user, two helping systems are present (marked by magenta dash-dotted line). An exhaustive mouse over hoover tooltip help that is popping up any relevant information for tools / buttons under the cursor. To not distract the user while working on a model, the tool tips are disabled for elements in the input area.

Additionally, there is a static help bar at the bottom, providing information about the element the cursor is currently pointing at. Advanced users can hide the static help using the toolbar.

For clients with a small screen, like mobile devices, the layout changes to only display the input or the output. The user can slide between those two sides using a slider on the side[13].

The input area was designed to show a moveable and zoomable image from the survey (using click and drag and the mouse wheel). Besides the zoom functionality which utilizes the mouse wheel, all other actions on the input area are triggered by simple click or click and drag, avoiding double clicks and any key combinations. This allows the input area to be easily used with devices featuring touch interface and allows for a more natural and easy-to-use UI [12, 1].

The user has to identify lensed images and order them in respect to arrival time. Input is restricted to assist the user in creating valid configurations. Users have to start with one point and expand the existing figure if necessary by clicking on existing points. While this seems counterintuitive and needs explanation[14], trials show that users get the grip rather quickly. The advantage of always creating logically valid input that is consistent with the theoretical background of gravitational lenses enhances the understanding.

The user has tree tools available, one is always active. The identification tool (i5) is used to mark the lensed images. The point mass tool (i4) is used to mark external point masses and the ruler tool (i6) allows to quickly estimate distances.

3.1.3 User Interface (UI) – Implementation

The user interface implementation (JS object `LMT.ui`) is split up in three parts `LMT.ui.html`, `LMT.ui.svg` and `LMT.ui.out`, as is the layout of the application.

All the user input for creating a model is handled by `LMT.ui.svg`, as is the display of the input pane. The input area is implemented as a dynamic SVG image. This SVG image gets created on the fly when `LMT.ui.svg.init()` is called and it's DOM is directly embedded in the HTML DOM.

The SVG DOM is organized as a tree of layers, where UI elements get painted on. The root layer containing all the others defines the visible area by setting the `transform` attribute. The root layer has four child layers, the first containing the background raster image. The second contains all the SVG elements representing the point masses. The third contains all elements representing the model, like extremal points, contour lines and so on. The last is used to draw the temporary UI elements for the rulers.

The background image(s), once loaded by `LMT.com` during the start up, are handled by `LMT.ui.svg.bg`. They are first rendered onto an invisible HTML `canvas`. On this canvas, pixel based image operations can be applied to the input image. This step allows adjusting brightness, contrast and the composing of multiple input images (originating from multiple band filter) in the future. As soon as the canvas has finished rendering the

[13]Note that mobiles are currently not supported due to the lack of testing devices.
[14]A tutorial is needed anyways

images, the composed image data is extracted using the `.toDataURL()` function. The resulting object is then put in the SVG DOM, on the background layer as a `SVGimage`.

All other layers are populated by their according JS object instances from the model, saved in `LMT.Model` (consult Section 3.1.4). Each existing instance creates, manages and destroys it's visual representation, the according SVG DOM elements, themselves and place it on the SVG layer.

User event handling is implemented by `LMT.ui.svg.events`. This event manager catches standard low level user interface events on the SVG image and fires the according high level application event.

The function `LMT.ui.svg.ConvertToPNG()` triggers a rendering pass of the canvas, resulting in a rasterized image file to be saved and used as a preview image for the model.

The output pane `LMT.ui.out` consists of a set of HTML5 canvases lying on top of each other. If a simulated model was received, the event `ReceivedSimulation[]` gets fired. This calls the event handler `LMT.ui.out.load()` that renders each received image to a separate canvas. Image adjustments (like brightness and contrast) are applied. The canvases are hidden by default, using CSS properties. The event `DisplayOutputSlide[Nr]` triggers `LMT.ui.out.show()`. The canvas to be shown is first hidden, then placed on top of all canvases using the CSS `zIndex` property. Finally, jQuery is used to fade in this canvas.

General user interface actions, like showing dialog screens or tooltips, are handled by `LMT.ui.html`. Standard low level events like `onClick[]` on buttons trigger according high level events. The module `LMT.ui.html` makes use of the features of jQuery[15], in order to show and to style the elements in the HTML DOM that represent the dialog boxes, buttons and so on.

3.1.4 Data model and application state

The program is always in a state, defined through the attributes of all the objects used for data storage. In SpaghettiLens, the data storage is distributed among several objects. The state of the application can be split up in the state of the model, and the state of the UI.

The state of the model is saved in `LMT.Model`, an instance of type `LMT.objects.model`. The model `LMT.Model` consists of an array of sources, an array of external masses (instances of `LMT.objects.ExternalMass`) and a plain object `Parameters`. Each source is represented by a binary tree structure[16] of instances of `ExtremalPoint` (`LMT.objects.ExtremalPoint`). ExtremalPoints are of a certain type, minima (`"min"`), maxima (`"max"`) or saddle point (`"sad"`). They can be a leaf node of the tree (and thus either represent a `"min"` or `"max"`) or they can be the parent of exactly two child nodes (and therefore be a `"sad"` by definition) and have two child ExtremalPoints. The root node of this tree is always a `"min"`. If two child nodes come close together, they and

[15]and it's plugin jQuery UI

[16]Binary tree: each parent element can have two children

their parent represent a limaçon configuration, where the two child nodes are of opposite type. Otherwise it is a lemniscate configuration[17], where the child nodes are of the same type. The angle that triggers the switching from limaçon to lemniscate is stored in `LMT.model.Parameters.MinMaxSwitchAngle`.

This tree structure guarantees that the odd number theorem is satisfied. It allows a iteratively refined by adding smaller order perturbations as you expand the tree structure downwards.

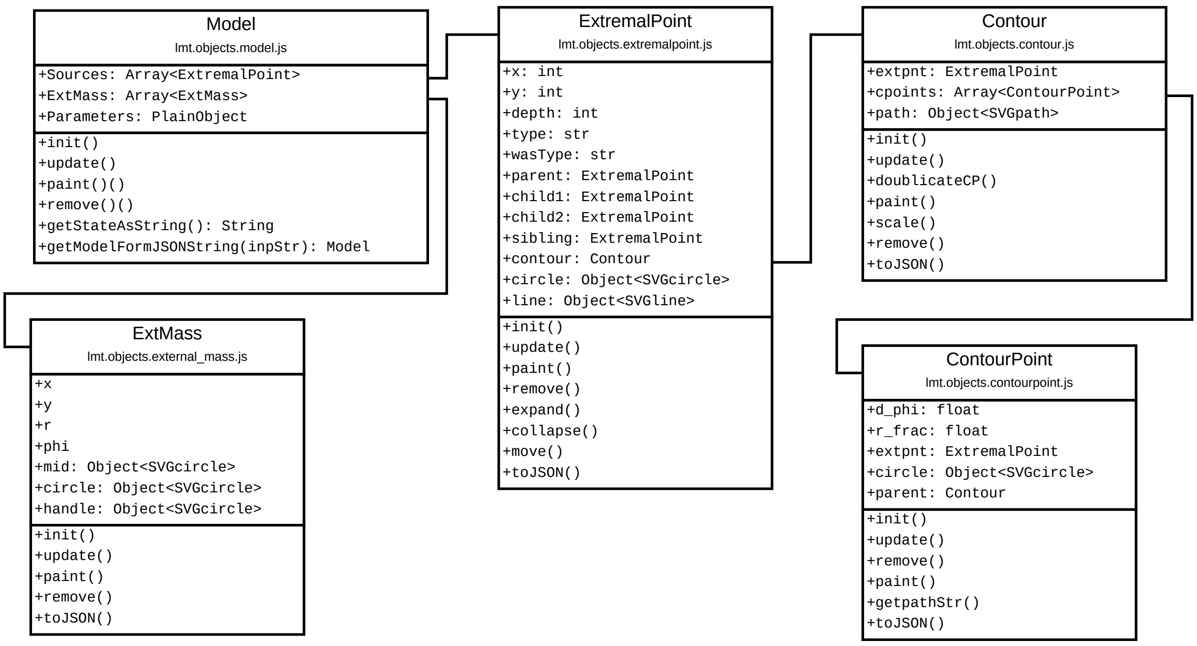


Figure 5: Class diagram of a model. Only important attributes and methods are shown.

For each `ExtremalPoint` (except the root node) only the inner most self intersecting, enclosing contour line is drawn on the UI. This contour line intersects itself at the parent saddle point and is one part of a limaçon or lemniscate curve. Contour lines are instances of `LMT.objects.contour` and are implemented as a sequence of bézier curves. A contour consists of contour points (instances of `LMT.objects.contourpoint`), a variable number of points that represent end points of the bézier curve segments.

UML Diagram 5 gives an overview of all objects representing a model.

The state of the model (and all it's dependencies) can be parsed to a JSON string using the models `getStateAsString()` method. Vice versa, such a string can be converted to a model using the abstract method `LMT.objects.model.getModelFronJSONStr(str)`, followed by a call to `update()` and `paint()` to initialize the internal data structures and paint the model on the SVG interface. This mechanism is used to save model states in the action stack for undo and redo purposes, and to send them to the server side for simulation. The Listings 3.2 and 3.3 show the elements of which this JSON string is composed in a structured notation.

[17]figure of eight

```
{
  "__type":"model",
  "NrOf": NrOfObj,
  "Sources": SourcesArray,
  "ExternalMasses": ExternalMassesArray
  "Rulers":[],
  "MinMmaxSwitchAngle": <float>,
  "Parameters": ParametersObj
}
//---------------------------------------------------
NrOfObj: {
  "__type":"counters",
  "Sources": <int>,
  "ExtremalPoints": <int>,
  "Contours": <int>,
  "ContourPoints": <int>
}
//---------------------------------------------------
"Parameters": {
  "pixrad": <int>,
  "n_models": <int>,
  "isSym": <Boolean>,
  "z_src": <int>,
  "z_lens": <float>,
  "orgPxScale": <float>,
  "orgImgSize": <int>,
  "svgViewportSize": <int>,
  "pxScale": <float>
}
//---------------------------------------------------
ExternalMassesArray: [ ExtMass1, ExtMass2, ...]
ExtMassX: {
  "__type":"ext_mass",
  "idnr": <int>,
  "x": <float>,
  "y": <float>,
  "r": <float>,
  "phi": <float>,
}
//---------------------------------------------------
```

Listing 3.2: The building blocks of a model in JSON notation (page 1 of 2)

```
SourcesArray: [ ExtPnt1, ExtPnt2, ... ]
ExtPntX: {
  "__type":"extpnt",
  "idnr": <int>,
  "x": <float>,
  "y": <float>,
  "depth": <int>,
  "isRoot": <Boolean>,
  "isExpanded": <Boolean>,
  "childrenInsideEachOther": <Boolean>,
  "type": <"sad"|"min"|"max">,
  "wasType": <"sad"|"min"|"max">,
  "child1": ExtPntX1,
  "child2": ExtPntX2,
  "contour": ContourXY
}
//----------------------------------------------
ContourXY: {
  "__type":"contour",
  "idnr": <int>,
  "cpoints": [ CPntXY1, CPntXY2, ... ]
}
//----------------------------------------------
CPntXYZ: {
  "__type":"cpnt",
  "idnr": <int>,
  "r_fac": <float>,
  "d_phi": <float>
}
```

Listing 3.3: The building blocks of a model in JSON notation (page 2 of 2)

Besides the action stack, implemented in `LMT.actionstack` as an instance of `LMT.objects.ActionStack`, there are further objects representing the state of the UI. The object `LMT.datasource` keeps track of information related to the origin of the lens image. The object `LMT.modelData` saves the server side database entry for the current lens. The database entries for a simulated model (`ModellingResult`) are saved in `LMT.simulationResult`. General settings influencing the appearance of the UI not connected to the modeling process are stored in `LMT.settings`.

3.1.5 Communication Interface (COM)

The communication with the server is handled by the `LMT.com` object. This is a collection of event handler functions that send and get data to the server in the background using the interface described in Section 3.2. Upon receiving data, it is stored and afterward the app is notified by firing the according event.

If the client expects data from the server, a polling loop is started in the background, as described in Section 2.4.5.

3.1.6 Data sources modules

SpaghettiLens can load data from different data sources. Those data sources are implemented as member objects of the object `LMT.datasources`. Each data source object has a corresponding server side class. Data source objects are initialized if selected in the initial dialog. They are supposed to populate `LMT.datasource` with their data and can make use of `LMT.ui.html.GenericDatasourceDialog` that gets designed by the server side template.

They have to register an event handler for `LensesSelected[]`. This event gets fired, when the user clicks on OK in the custom `GenericDatasourceDialog`. The event handler can be used to determine the database model id in combination with the server side part of the data source module (by either creating a new database entry or returning an existing one). This communication bypasses the `LMT.com` object and uses its own api `datasourceApi`. Once the id of the model to be loaded is determined, the data source module is expected to return control to the main app by firing `GetModelData[models]`, where `"models"` is an array of model ids to load.

Alternatively, this process can be skipped by starting up the web application with a known, existing model id using `GET` parameter `mid`[18] or by loading an existing result using the result id `GET` parameter `rid`[19].

This rather complicated setup of client and server side data source modules allows to circumvent any "same origin policy" issues that can not be solved using CORS, as explained in Section 2.4.5.

[18]`http://mite.physik.uzh.ch/?mid=42`
[19]`http://mite.physik.uzh.ch/?rid=1337`

3.2 Interface Client Server

This section explains the protocol used for the communication between the client and the sever application. Technically, this protocol uses HTTP to send JSON strings with defined content. All requests are supposed to return immediately, as the original HTTP specification requests. Server to client communication is achieved using polling.

The server offers three points for the exchange of data: `/api/`, `/tools/` and `/data/`. Since there is some legacy code still in the code base, there are additional, depreciated url access points to exchange data (`/get_initdata/`, `/get_modeldata/`, `/save_model/`, `/save_model_final/`, `/load_model/`, `/result/`). UML Diagram 6 depicts an overview of all access points, their supported HTTP request types and the according client side functions that access this points. Whereas requests to `/api/` should never be cached, the other interfaces represent static URI to static content and should be aggressively cached, once available. This implements the idea of a RESTful API as proposed by [8].

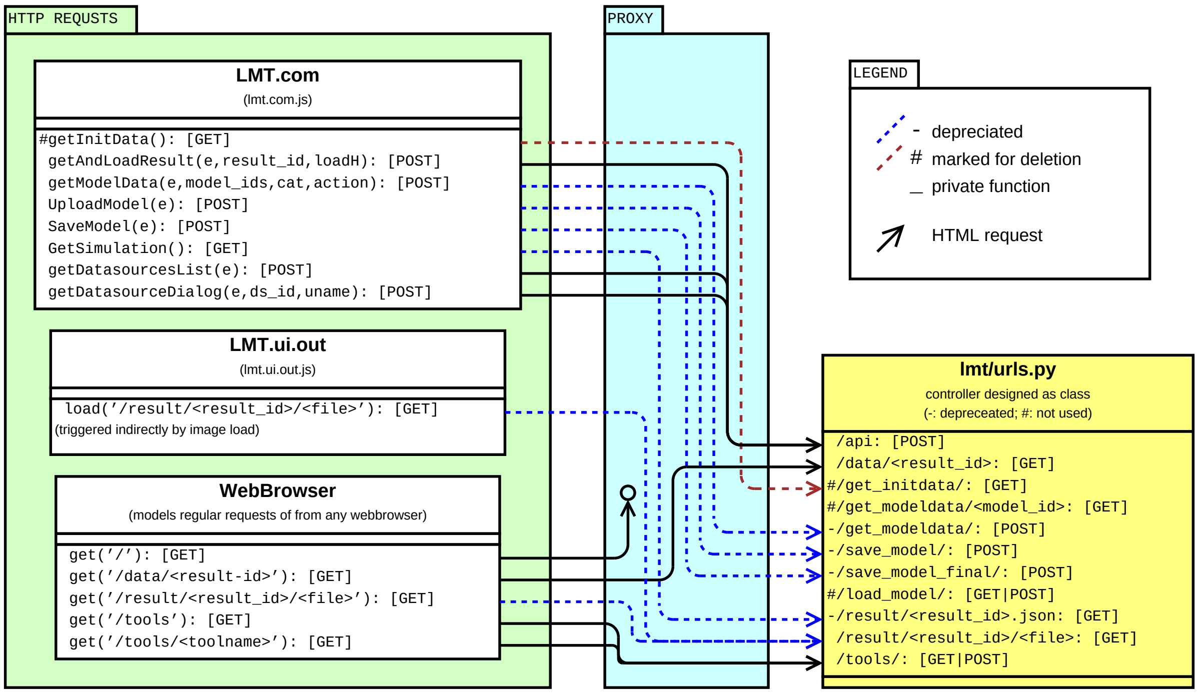

HTTP REQUSTS
LMT.com
(lmt.com.js)
#getInitData(): [GET]
getAndLoadResult(e,result_id,loadH): [POST]
getModelData(e,model_ids,cat,action): [POST]
UploadModel(e): [POST]
SaveModel(e): [POST]
GetSimulation(): [GET]
getDatasourcesList(e): [POST]
getDatasourceDialog(e,ds_id,uname): [POST]
LMT.ui.out
(lmt.ui.out.js)
load('/result/<result_id>/<file>'): [GET]
(triggered indirectly by image load)
WebBrowser
(models regular requests of from any webbrowser)
get('/'): [GET]
get('/data/<result-id>'): [GET]
get('/result/<result_id>/<file>'): [GET]
get('/tools'): [GET]
get('/tools/<toolname>'): [GET]
PROXY
LEGEND
- depreciated
# marked for deletion
_ private function
HTML request
lmt/urls.py
controller designed as class
(-: deprecated; #: not used)
/api: [POST]
/data/<result_id>: [GET]
#/get_initdata/: [GET]
#/get_modeldata/<model_id>: [GET]
-/get_modeldata/: [POST]
-/save_model/: [POST]
-/save_model_final/: [POST]
#/load_model/: [GET|POST]
-/result/<result_id>.json: [GET]
/result/<result_id>/<file>: [GET]
/tools/: [GET|POST]

Figure 6: An overview over the interface client – server. Shows the origin of requests with green background. The caching proxy with blue background. Entry points / controller definitions with yellow background. (no valid UML notation)

The interface `/api/` is supposed to handle all bidirectional, dynamic communication with the client application. Clients are required to do HTTP POST requests to `http://mite.physik.uzh.ch/api` having a JSON string as message body. The JSON object is required to have at least the parameter `"action"` set to the value of the desired action. On the server side, the value of the action field is evaluated and the corresponding server side function is called, using the other fields as key value arguments for the function. The source code files `/backend/ModellerApp/views.py`, *`lmt.com.js`* and UML Diagram 7 provide further details about the implementation.

A common task, the upload of a model, requires the JS object “model” to be converted to a JSON string. Listings 3.2 to 3.3 in Section 3.1.4 shows the JSON communication format for sending models between client and server in a structured notation. This has the same tree–like structure, as is shown in UML Diagram 5.

The access point `/data/` presents results of the simulation of a model. It answers to a HTTP GET request by parsing the rest of the URL to an integer and returning a web page containing all the results for that result number. It can be used from any web browser, visiting the page `http://mite.physik.uzh.ch/<result-id>`[20]. The result of this request is a simple HTML page, offering an overview of all the data of this particular model.

`/tools/` offers access for advanced users and scientists to tools used for administering collaborative modeling, getting more detailed data and so on. At the moment, only one tool is implemented: `/tools/ResultDataTable/`. This tool creates an overview table over all parameters for a set of result ids. The resulting table can be downloaded as a Excel file (`csv`) or directly shown in a browser (`HTML-table`). The query to the data base is directly composed of the HTTP GET parameters: `http://mite.physik.uzh.ch/tools/ResultDataTable?6696,6904-7000&type=html`. For a full documentation visit the tool without any argument[21]. Further tools will be developed and implemented in cooperation with volunteers and scientists.

[20]example: `http://mite.physik.uzh.ch/1337`
[21]`http://mite.physik.uzh.ch/tools/ResultDataTable`

3.3 Proxy

In the current setup, there is a proxy server between the application server and the internet. It's purpose is to reduce the load of the application server and prevent the firing up of a resource intensive[22] python thread in the app server on every request.

It is supposed to directly serve the client application. Further, this particular server is set up to serve `labs.spacewarps.com` and the SpaghettiLens documentation and tutorial page in combination with a php process.[23]

It is setup to further intercept requests to `/data/` and return the data directly, if available. Only if the result first needs to be rendered, it is passed on to the application server.

It is also set up to cache any requests made to `/api/` that can be regarded as static, like getting the information about a lens.

At the moment, the proxy uses the web server software nginX ("Engine X") that is preferred over Apache because of the small memory finger print. This is due to the fact that nginX is event based, in opposite to Apache, which is process based. nginX is often used as proxy server and load balancer to serve static content infront of other server like apache that handle the dynamic content.

[22]CPU time and memory size
[23]this part is still work in progress

3.4 Server-side Scripts (Server)

As explained in Section 2.4.2, the server application is basically an interface to the database. SpaghettiLens splits the server-side scripts further up and distinguishes between the module that manages data (server) and the part that generates data (worker) that is introduced in Section 3.5.

The server manages all the data needed and generated by the client app. It keeps track of lenses that can be modeled (`LensData` objects identified by `model_id`[24]), create new ones from the data sources available, created models (`ModellingResult` objects, identified by `result_id`) and keeps track of simulation results and files.

The server provides the client unified access to survey images hosted on several possible external data sources (at the moment, SpaceWarps is fully and MasterLens partially implemented). It keeps track which models (results) are for which lens, which user created a model and which models are descendants from each other.

Additionally, it is the interface between the client app and the worker that do the actual modeling. It has to convert the data generated by the client (JSON) to a format that the modeling application requires (GLASS config files).

3.4.1 Techniques used

The server is implemented in Python language, using the Django framework. The framework is served using gunicorn, a WSGI HTTP server implemented in Python.

The Django framework follows a model-view-controller architecture. It defines a data model which is represented in a back end database. A view is a representation of a subset of this data presented to a user (the client app in this case). The controller allows to modify the data.

The data model is created by defining Python classes in `/backend/ModellerApp/models.py`. Django uses a relational database in the back end, to keep track of all the created instances of those models. For the actual storage of the data, Django relies on a external relational database. For SpaghettiLens, MySQL, the most popular[14] open source database, was chosen.

Views and controllers are defined by Python functions (`/backend/ModellerApp/views.py`). Those functions get mapped to HTTP requests by `/backend/lmt/urls.py`. They process a request and either query the model / database to return a representation of the data or update the data.

Django offers a powerful template system that allows to render the data in any possible form. Usually it's used to render HTML pages, but this setup uses it to return JSON data obejcts to the client app. The template system is used to render the result views[25], which can be accessed directly by a browser to display generated models.

To simulate the models, the server needs to connect to the simulation software (GLASS), tell it what to do and fire it up. The server creates a config file for GLASS for each request, and then starts an instance of GLASS running this file. Since the simulation takes

[24]Will be renamed to `lens_id`.

[25]`http://mite.physik.uzh.ch/data/<result_id>`

in the order of minutes, the server worker process should not be blocked during simulation. Additionally, this part should be easily scalable, since it creates the biggest server load and needs to be up scaled linearly with the amount of users that use SpaghettiLens concurrently. This is done using a task distribution system (Celery) that gets introduced in Section 3.5.

3.4.2 The Design of the API

UML Diagram 7 shows a complete overview of the server API. Note that the API is currently under redesign. The old design paradigm stated that each possible action of the server was exposed direcly to an url, where a client could make a HTTP request to. This assignment is confiured in `/backend/lmt/urls.py`. It has the draw back that those access points need to be configured on the server side in two places, in the server application, and additionally in the proxy.

The new design paradigm, there are only three access points offered. this simplifies the administration of the proxy server and increases the flexibility of the application.

3.4.3 The Design of the Database

The database schema is shown in UML Diagram 8. There are two important tables in the database. The first is the table of all lenses that have been modeled, `LensData`. It stores all the objects from all different data sources, as soon as they have been requested once. It keeps track of the origin of the image in the field `datasource`. The location of the image files is stored in `img_data`, as a JSON string. This JSON object needs at least a key `url`, pointing to a publically available image. If an lens consists of multiple images (multiple filters / bands...), those can be stored in this JSON object too.[26] Arbitrary additional data (for use in the corresponding datasource modules client and server side) can be stored as JSON strings in `add_data`.

The second table, `ModellingResult`, keeps track of all the models that were generated. It keeps track of the `LensData`, it is a result of and stores the JSON string generated by the client that represents this model in `json_str`. If this result is a refined version of a previous, the key of the previous is saved in `parent_result`, creating a tree like structure. Additional administration information is kept in `created_by`, `rendered_last`, `last_accessed`. This information can be used for later clean up purposes as the deletion of the modeling state files and rendered images for models that are not visited anymore. GLASS parameters used to generate the model are stored in the according fields.

3.4.4 The parsing of a model JSON string

In order to save a received model from the client app, the server parses the JSON string received and converts it to a temporarily python data structure. This is done in `/backend/ModellerApp/utils.py`, using the class `EvalAndSaveJSON`.

[26]to be implemented in the client side.

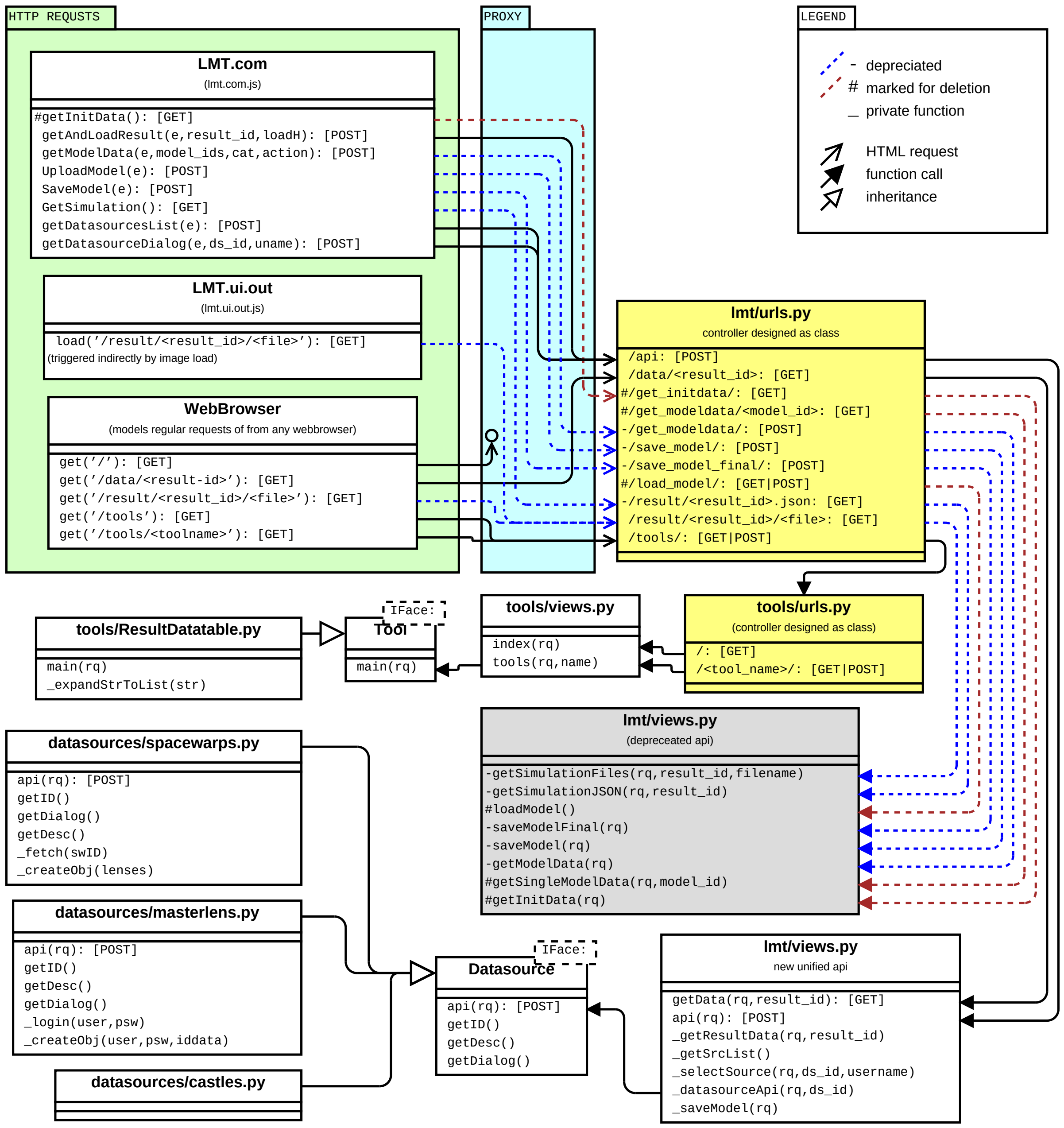
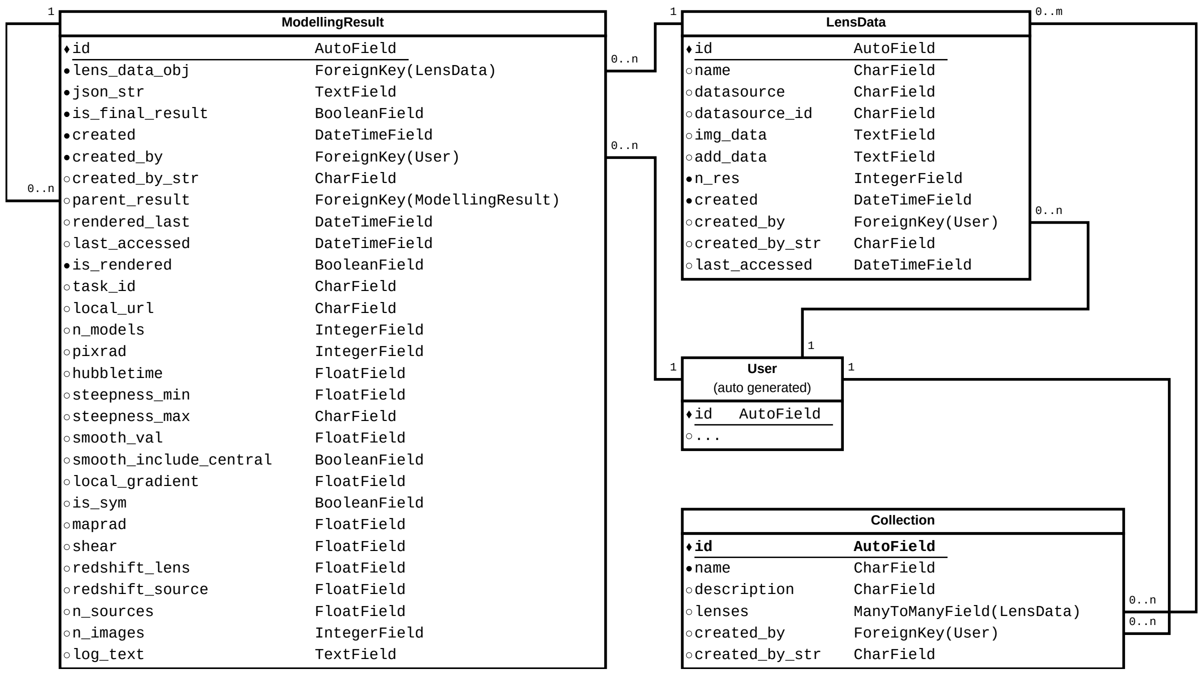


Figure 7: An overview over the server API and the internal functions called. Shows the origin of requests with green background. The caching proxy has a blue background. Entry points / controller definitions with yellow background. Server list of server functions has a white background. Lines show flow of data (function calls, HTML requests). Dashed blue and red lines show function calls and grey functions will be removed in future versions (no valid UML notation)

`EvalAndSaveJSON.__init__()` first sets all the default values for glass and then calls `EvalAndSaveJSON.evalModelString()`. This function parses the JSON model string to a python dict containing Python objects. The parameters of the uploaded model are



ModellingResult	
♦id	AutoField
•lens_data_obj	ForeignKey(LensData)
•json_str	TextField
•is_final_result	BooleanField
•created	DateTimeField
•created_by	ForeignKey(User)
○created_by_str	CharField
○parent_result	ForeignKey(ModellingResult)
○rendered_last	DateTimeField
○last_accessed	DateTimeField
•is_rendered	BooleanField
○task_id	CharField
○local_url	CharField
○n_models	IntegerField
○pixrad	IntegerField
○hubbletime	FloatField
○steepness_min	FloatField
○steepness_max	CharField
○smooth_val	FloatField
○smooth_include_central	BooleanField
○local_gradient	FloatField
○is_sym	BooleanField
○maprad	FloatField
○shear	FloatField
○redshift_lens	FloatField
○redshift_source	FloatField
○n_sources	FloatField
○n_images	IntegerField
○log_text	TextField

LensData	
♦id	AutoField
○name	CharField
○datasource	CharField
○datasource_id	CharField
○img_data	TextField
○add_data	TextField
•n_res	IntegerField
•created	DateTimeField
○created_by	ForeignKey(User)
○created_by_str	CharField
○last_accessed	DateTimeField

User (auto generated)	
♦id	AutoField
○...	

Collection	
♦**id**	**AutoField**
•name	CharField
○description	CharField
○lenses	ManyToManyField(LensData)
○created_by	ForeignKey(User)
○created_by_str	CharField

Figure 8: UML database schema. Using Django notation for field types.

then, after a sanity check, written over the default values.

In a second step, `EvalAndSaveJSON.orderPoints2()` reorders the tree like structure of extremal points provided by the client to a list of images ordered by their expected arrival time. It determins the center of the lensing galaxy (assumed to be the maximum) This also converts all the measurements from SpaghettiLens pixels to actual arcsecs and makes them reative to the center of the lens.

The third step `EvalAndSaveJSON.createModellingResult()` creates the database entry in `ModellingResult`. The last step `EvalAndSaveJSON.createConfigFile()` creates the GLASS config file to be supplied to the modeling software.

```
# LMT_GLS_v4
# LMT_v1.6.3
import matplotlib as mpl
import pylab as pl
glass_basis('glass.basis.pixels', solver='rwalk')
meta(author='c_cld', notes='using LensModellingTools')
setup_log('../tmp_media/009719/log.txt')
samplex_random_seed(0)
samplex_acceptance(rate=0.25, tol=0.15)
exclude_all_priors()
include_prior(
  'lens_eq',
  'time_delay',
  'profile_steepness',
  'J3gradient',
  'magnification',
  'hubble_constant',
  'PLsmoothness3',
  'shared_h',
  'external_shear',
)
hubble_time(13.700000)
globject('9719__ASW0007j6j')
zlens(0.500)
pixrad(8)
steepness(0,None)
smooth(2.00,include_central_pixel=False)
local_gradient(45.00)
symm()

shear(0.01)

A = 1.833, 0.733
B = -0.197, -1.805
source(1.000,
  A, 'min',
  B, 'sad', None)
model(200)
savestate('../tmp_media/009719/state.txt')
env().make_ensemble_average()
```

Listing 3.4: GLASS config file (page 1 of 2)

```
env().arrival_plot(env().ensemble_average, only_contours=True,
                   colors='magenta', clevels=40)
env().overlay_input_points(env().ensemble_average)
pl.gca().axes.get_xaxis().set_visible(False)
pl.gca().axes.get_yaxis().set_visible(False)
pl.savefig('../tmp_media/009719/img1.png')
pl.close()
env().kappa_plot(env().ensemble_average, 0, with_contours=True,
                 clevels=20, vmax=1, with_colorbar=False)
pl.gca().axes.get_xaxis().set_visible(False)
pl.gca().axes.get_yaxis().set_visible(False)
pl.savefig('../tmp_media/009719/img2.png')
pl.close()
env().srcdiff_plot(env().ensemble_average)
env().overlay_input_points(env().ensemble_average)
pl.gca().axes.get_xaxis().set_visible(False)
pl.gca().axes.get_yaxis().set_visible(False)
pl.savefig('../tmp_media/009719/img3.png')
pl.close()
env().srcdiff_plot_adv(env().ensemble_average, night=True, upsample=8)
env().overlay_input_points(env().ensemble_average)
pl.savefig('../tmp_media/009719/img3_ipol.png', facecolor='black',
           edgecolor='none')
pl.close()
LMT={
 'svgViewport' : 500,
 'orgImgSize'  : 440,
 'pxScale'     : 0.16456,
 'orgPxScale'  : 0.18700,
 'gls_version' : 'v4',
 'lmt_version' : 'v1.6.3',
}
```

Listing 3.5: GLASS config file (page 2 of 2)

3.5 Worker

The actual simulation of the models is done by GLASS. Since the modeling takes in the order of minutes, a modeling task can not simply be done by the server. Depending on the server architecture, this would block at least a web server worker thread, making it unable to respond to other requests. Since the simulation of models is the most CPU intensive task, this needs to be easy scalable, in case more users start to use SpaghettiLens. Preferably, the simulation jobs could be distributed to several machines.

GLASS is a command line application that expects a config file to run on.

For the management of the worker thread Celery was chosen.

3.5.1 Celery overview

Celery is distributed task queue implemented in Python. It offers integration packages for many web frameworks, including Django. This makes the setup of a distributed task system with Django and Celery easy.

```python
from lmt.tasks import calculateModel
from celery.result import AsyncResult

@csrf_exempt
def getSimulationJSON(request, result_id):
  # ...
  res = ModellingResult.objects.get(id=result_id)
  # ...

  # set time limits
  dt = 60*30 # task has 30min till it gets revoked
  expire = 60 # task gets canceled after 60s in queue

  task = calculateModel.apply_async(args=(result_id,), timeout=dt,
                                    expires=expire)
  res.is_rendered = False
  res.task_id = task.task_id
  res.rendered_last = now();
  res.save()
```

Listing 3.6: Example of how to call a task from Django. Simplified version of `getSimulationJSON()` from `/backend/ModellerApp/views.py`

Possible units of work, called tasks, can be defined as python function on the server side in Django using the file `/backend//lmt/tasks.py`. This tasks can be started by Django asynchronously by a call to the task as shown in Listing 3.6 and return a `AsyncResult` instance that allows to check the state of the task or it's return value in a next step.

Once a task is started by Django, Celery takes over. The task is sent to a message broker that implements the task queue. The celery worker threads then consume the tasks from this queue, process it and save the result back in a result back end. The result backend keeps also track of the tasks states.

All these processes communicate over TCP sockets and thus can run on different machines.

3.5.2 The tasks

```python
import time, os, subprocess
from django.conf import settings as s
from celery import task, current_task

if s.ROLE == "production_worker": #standalone worker?
  @task()
  def calculateModel(result_id):
    rq = current_task.request
    retval = subprocess.call(['../run_worker_glass',
                              '%06i' % result_id])
    return

else:  #worker running on server machine
  @task()
  def calculateModel(result_id):
    rq = current_task.request
    os.chdir(s.WORKER_DIR_FULL)
    retval = subprocess.call(['%s/run_glass' % s.WORKER_DIR_FULL,
                              '../tmp_media/%06i/cfg.gls' % result_id])
    return
```

Listing 3.7: Server side definition of worker task. Simplified version of `/backend/lmt/tasks.py`

The tasks are defined in `/backend/lmt/tasks.py` on the server side. It's a simple call for a shell script that gets the needed files, calls GLASS, and upload the results back to the servers file system. See Listing 3.7 for the definition of tasks and Listing 3.8 for the called shell file.

3.5.3 The message broker

The message brokering is done with RabbitMQ, the recommended task queue backend. It is open source software written in Erlang, implementing Advanced Message Queueing Protocoll (AMPQ).

```sh
#!/bin/sh
# this is run under /lmt/backend
cd ..
mkdir -p tmp_media/$1
wget -P tmp_media/$1 mite/result/$1/cfg.gls
cd worker
./run_glass -t 4 ../tmp_media/$1/cfg.gls
cd ..
# upload files to server
scp tmp_media/$1/* lmt@mite:/srv/lmt/tmp_media/$1/
rm tmp_media/$1/*
```

Listing 3.8: Shell script representing a task.

3.5.4 The worker threads

Worker threads are independent units that can be started on any machine that is able to connect to the message broker. They only need a local install that is a Python with installed Celery module. Thus it can run on any machine, even with non root access.

Since the worker thread fires up GLASS locally, it needs the config file generated by the server available. The files model state file and plots generated by the simulation then need to be uploaded to the servers media directory again. This is done using a shell script using `wget` for download and `scp` for upload, see Listing 3.8.

3.5.5 The result backend

Since the worker generates files as results that get uploaded to the file server by the task itself, a result backend is only needed to keep track of the states of the tasks. This implies that there is no big load on the result backend, and thus the same database as Django is using can be used, simplifying the setup. One just has to make sure that external workers not running on the server machine can reach the database server[27].

[27]Default MySQL configuration restricts access to connections from `localhost`

4 Program and Data Flow

This section gives an overview over the program and data flow over the major components for three non trivial cases using UML sequence diagrams. Sequence diagrams show which components of a application are active at a given time. They abstract internal processes in modules and depicts simplified function calls and return values.

4.1 Selection of a Data Source

The application start up involves two processes. The user first has to choose a data source, followed by selecting a lens to work on. This section shows the first process, until the display of the dialog to select a lens to work on. The next section elaborates the continuation.

In a first step, all static content like HTML, CSS and JS files are loaded from the server. This files are directly served by the proxy server. Internally, the client application now starts the initialization routines, as soon as the JS files are loaded.

In a next step, the client application needs to know, what data sources are implemented on the server, to display a dialog to the user to select from.

The last step is to display the data source specific dialog that allows the user to browse and select lenses that are in the data sources data base. The functionality of this dialog is implemented by the data source module selected and can vary among data sources. The data source module generates the HTML markup used to generate the dialog in the client application.

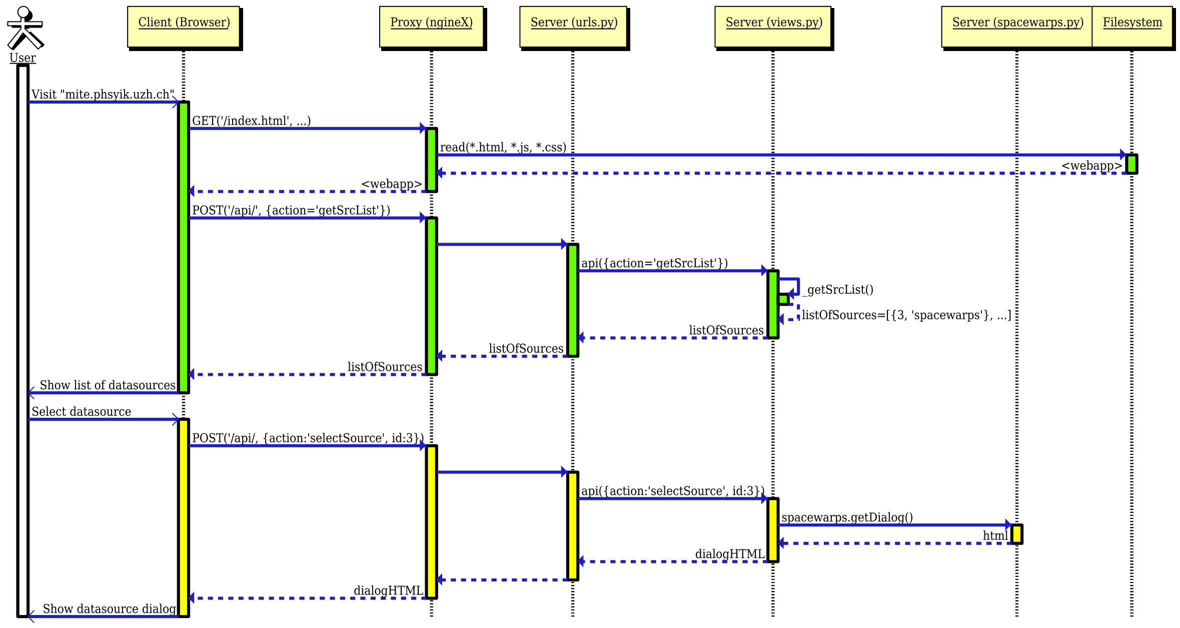


Figure 9: The two step initialization of SpaghettiLens. The first step loads the data selection dialog, followed by the data source specific lens selection dialog.

4.2 Selection of a Lens

The second step at the start up of the application is the selection of a lens to work on. This process differs depending on the selected data source in detail, but the overall idea is the same for all sources. Figure 10 shows this process for the SpaceWarps module. Note that the proxy server is not shown in this dialog, because it passes all the requests on to the server. Further, the server modules `/backend/urls.py` and `/backend/views.py` are combined, to simplify the diagram.

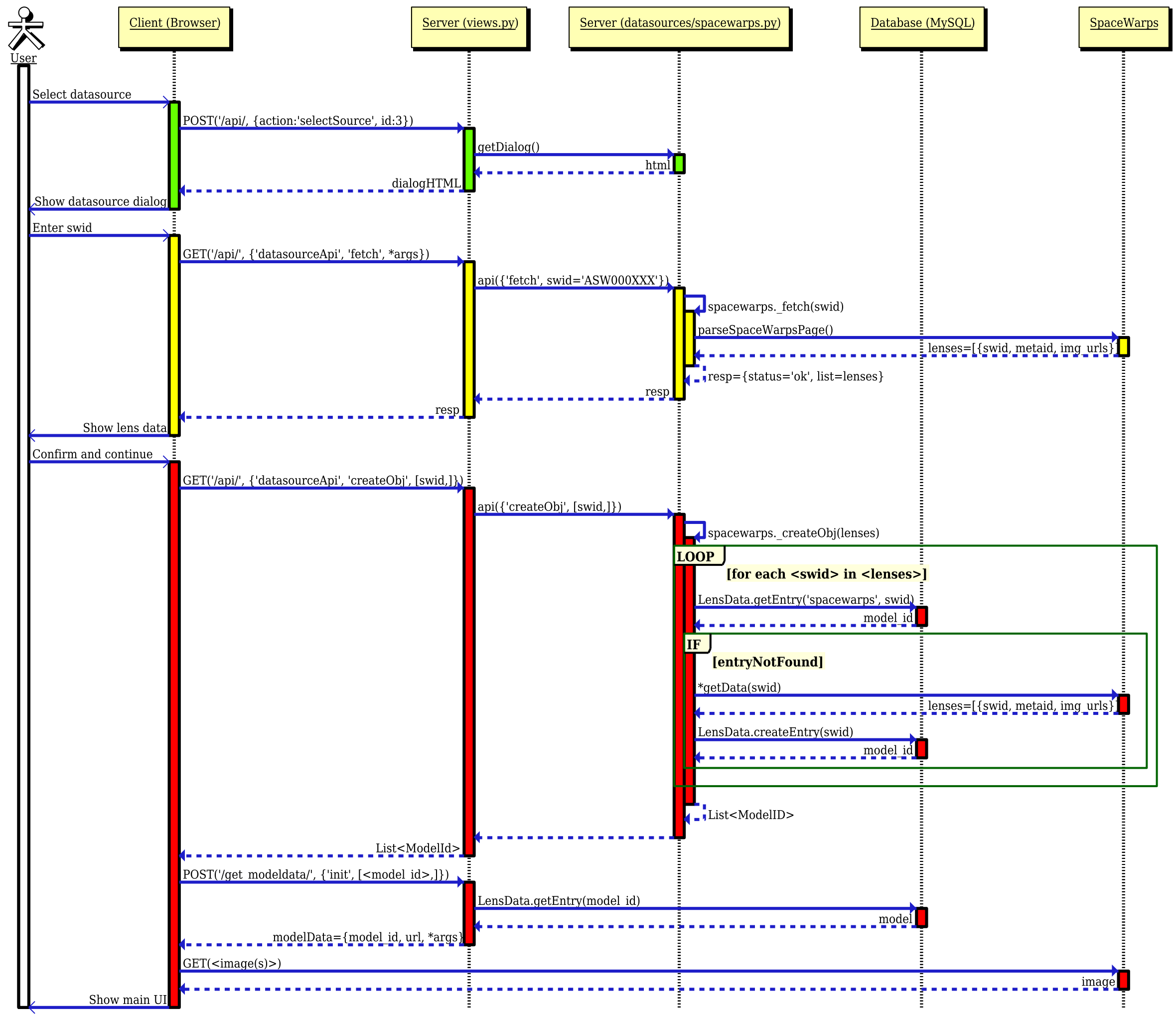

User
Client (Browser)
Server (views.py)
Server (datasources/spacewarps.py)
Database (MySQL)
SpaceWarps
Select datasource
POST('/api/, {action:'selectSource', id:3})
getDialog()
html
dialogHTML
Show datasource dialog
Enter swid
GET('/api/', {'datasourceApi', 'fetch', *args})
api({'fetch', swid='ASW000XXX'})
spacewarps._fetch(swid)
parseSpaceWarpsPage()
lenses=[{swid, metaid, img_urls}]
resp={status='ok', list=lenses}
resp
resp
Show lens data
Confirm and continue
GET('/api/', {'datasourceApi', 'createObj', [swid,]})
api({'createObj', [swid,]})
spacewarps._createObj(lenses)
LOOP
[for each <swid> in <lenses>]
LensData.getEntry('spacewarps', swid)
model_id
IF
[entryNotFound]
*getData(swid)
lenses=[{swid, metaid, img_urls}]
LensData.createEntry(swid)
model_id
List<ModelID>
List<ModelId>
POST('/get_modeldata/', {'init', [<model_id>,]})
LensData.getEntry(model_id)
model
modelData={model_id, url, *args}
GET(<image(s)>)
image
Show main UI

Figure 10: The selection and loading of a lens. First step (green) depicts the last action Figure 9; the selected id is validated in step two (yellow); validated lens data base entry is loaded or created if not available and the UI initialized (red).

It involves to offer the user a selection for a lens. This can be done for example using drop down fields or a simple text field. The data source module has to guarantee that this selection is valid. Otherwise, the user should not be able to continue.

The SpaceWarps data source offers a simple text filed for entering a SpaceWarps image id. It then checks the SpaceWarps data base, whether this id exists. If it does, it fetches additional information, displays it to the user and enables the button to continue. Otherwise a error message is shown. Figure 10 shows this part, colored in yellow.

The last step, colored in red in Figure 10, involves three parts. The data source module has to return a `model_id`, representing the selected lenses in the server side database. It has to guarantee the uniqueness of each lens in the database. It first needs to check if the selected lens(es) already exist in the database. If it does, this id is returned. Otherwise a new entry is creating and returned.

In the second part, the main part of the client application now queries the data base for the lens to be shown next. The response includes at least one URL to the actual image that gets loaded in the third part and then rendered to the screen.

This concludes the start up procedure and hides all dialog pop ups, allowing the user to begin modeling.

Alternatively, the user could load SpaghettiLens passing a GET argument `mid` or `rid`. This start up skips all the parts involving the data source modules and directly queries the server data base for the lens information and loads it. If a result is loaded using `rid`, an additional query to the data base is done, returning the JSON string representing the model. After loading the lens data, the model data is loaded and shown.

4.3 Simulation

After the user created a model, it can be sent to the server to get simulated and once finished, retrieve the resulting figures. Figure 11 shows the whole process.

In a first step, the model JSON string is parsed, validated and evaluated. It is saved to the data base as a new entry and the resulting id is returned to the client. Additionally, a configuration file for the simulation back end is written to disk.

Then the client asks for the location of the results of this model. Since there are no results yet, the server creates a new simulation task and hands it over to the broker that puts it in the task queue (Figure 11, yellow). A reference to this task and the fact that the simulation has started is saved in the data base and sent to the client.

As soon as a worker is idle, it consumes the next task in the queue and executes it (Figure 11, red). A task consist of getting the configuration file, running GLASS, producing the output figures and uploading those back to the server file system. The worker keeps track of the state of the task in the database.

Meanwhile, the client repeatedly checks with the server, whether the task has been completed. If so, the server sends back links to the generated figures that the client then loads and displays to the user.

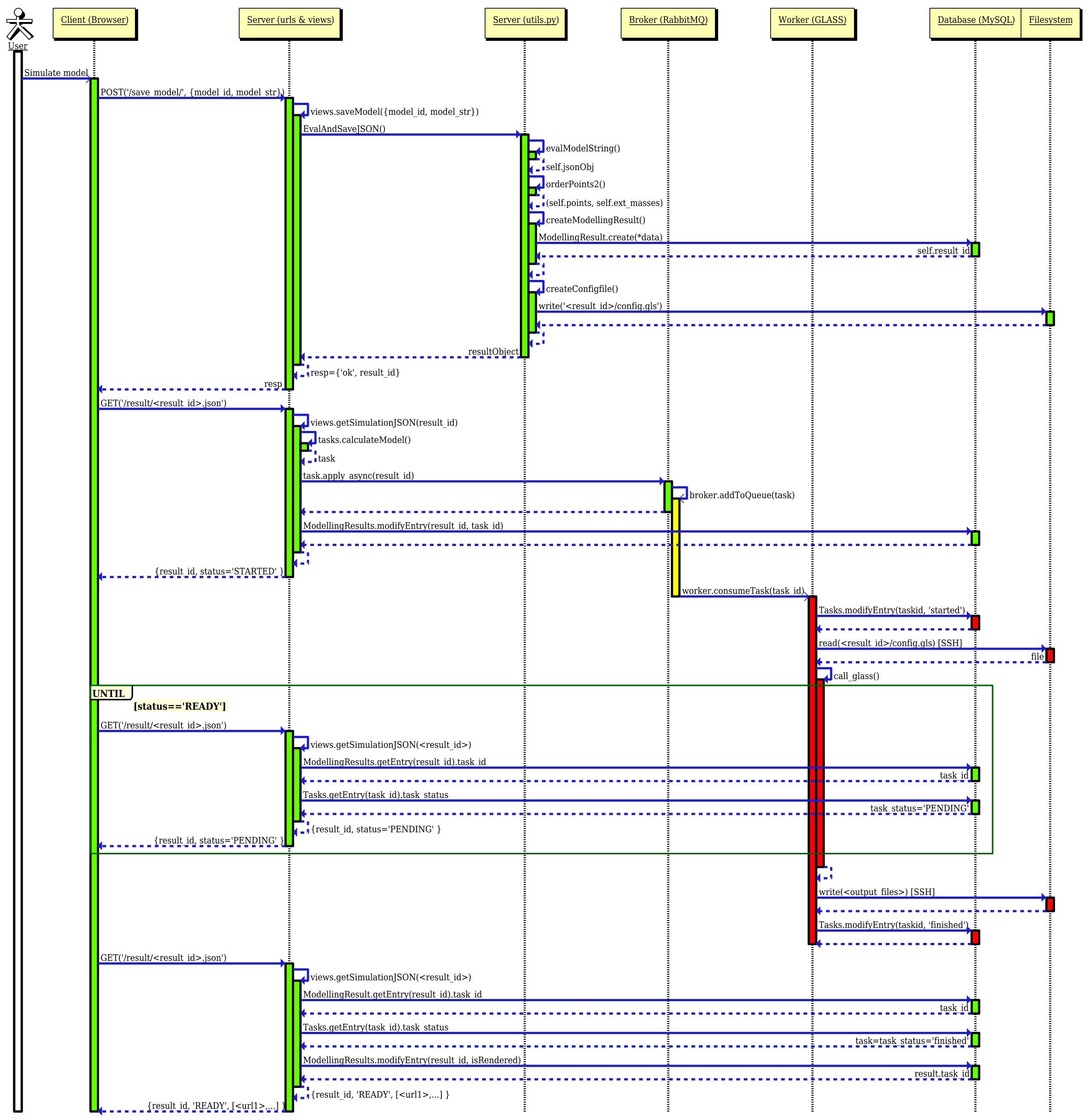

User
Client (Browser)
Server (urls & views)
Server (utils.py)
Broker (RabbitMQ)
Worker (GLASS)
Database (MySQL)
Filesystem
Simulate model
POST('/save_model/', {model_id, model_str})
views.saveModel({model_id, model_str})
EvalAndSaveJSON()
evalModelString()
self.jsonObj
orderPoints2()
(self.points, self.ext_masses)
createModellingResult()
ModellingResult.create(*data)
self.result_id
createConfigfile()
write('<result_id>/config.gls')
resultObject
resp={'ok', result_id}
resp
GET('/result/<result_id>.json')
views.getSimulationJSON(result_id)
tasks.calculateModel()
task
task.apply_async(result_id)
broker.addToQueue(task)
ModellingResults.modifyEntry(result_id, task_id)
{result_id, status='STARTED' }
worker.consumeTask(task_id)
Tasks.modifyEntry(taskid, 'started')
read(<result_id>/config.gls) [SSH]
file
call_glass()
UNTIL
[status=='READY']
GET('/result/<result_id>.json')
views.getSimulationJSON(<result_id>)
ModellingResults.getEntry(result_id).task_id
task_id
Tasks.getEntry(task_id).task_status
task_status='PENDING'
{result_id, status='PENDING' }
{result_id, status='PENDING' }
write(<output_files>) [SSH]
Tasks.modifyEntry(taskid, 'finished')
GET('/result/<result_id>.json')
views.getSimulationJSON(<result_id>)
ModellingResult.getEntry(result_id).task_id
task_id
Tasks.getEntry(task_id).task_status
task=task_status='finished'
ModellingResults.modifyEntry(result_id, isRendered)
result.task_id
{result_id, 'READY', [<url1>,...] }
{result_id, 'READY', [<url1>,...] }

Figure 11: Asynchronous simulation of a model. Shows the client polling for results until they are ready (green); the task queue (yellow) and worker process executing a task (red).

5 Configuration and Deployment

This section gives an overview of the possible setups and configuration of SpaghettiLens.

5.1 Configuration Files

All relevant config files can be found in the `./backend/settings` directory.

- **`base_settings.py`**
  This file contains the basic configuration of Django and internal Django setup. This should stay the same for all possible configurations.
- **`gunicorn.py`**
  This is the configuration file for the WSGI server running Django. Only used for the full server setup.
- **`machine.py`**
  This file sets up machine specific paths, the database connection and URLS. The `ROLE` parameter defines what kind of installation this setup is and which modules should be used.
- **`secrets.py`**
  This file contains information that should be kept secret like user credentials for the database.
- **`settings.py`**
  This is the master settings file that import all others. There are no parts to be changed by the user. It defines all possible roles.
- **`version.py`**
  Keeps track of version numbers. Is automatically updated by the install scripts.

5.2 Full Server – Installation

The full server installation installs all components on one machine that is supposed to serve as web server. Worker threads can additionally be run on different machines, see Section 5.4.

Due to the complicated setup, this process is automated with a interactive installation script. The installation script is tested on a fresh install of Ubuntu server edition, but should work with all Debian based systems. It needs a lot of packages, modules and configuration files to be setup and will create and change some system configuration files. A full install on a fresh installed OS will need about 1GB[28].

Warning: The install script will write and change configuration files for MySQL, nginX and RabbitMQ. If you have any of those running, please don't use the install script.

[28]Due to the many packages that need to be installed. Most systems already have most available.

Requirements for the install scripts are a installed Python 2.7 with packet manager PiP and the packages numpy, scipy, matplotlib and fabric. Additionally, you have to get a copy of GLASS.

The script is started by running the following command in the base directory:

```
$ fab install
```

All components should start up automatically on restart and SpaghettiLens should be available.

5.3 Full Server – Update

To update the server, first change to the source directory and make sure to have the latest version. The update scripts will copy the new files to the according locations and minify the HTML, CSS and JS files into single files.

```
$ fab update_backend:install_dir='/path/to/install'
$ fab update_html:install_dir='/path/to/install'
```

If there were updates to the database design, the new schema will have to be manually applied. SpaghettiLens uses the Python module south to help with schema migrations.

5.4 Worker Installation

Additional worker nodes can easily be spawned on any machine that has access to the file system of the server using SSH and can access the database. No root access is needed on the local machine. A step-by-step documentation of the installation is available in the file `./install/roles/production_worker.py`

5.5 Other Installations

If requested, the installation script can be modified to allow further installation types. The installation script lets the user select, what type of install is desired. Depending on the selection, it installs and configures the depended software.

This could easily be extended to support an installation on a local machine for private use. Refer to the files in `./install/` for further information.

6 Conclusions and Outlook

This section concludes with a (personal) evaluation of the development process, it states a few open problems still to be fixed and feature requests by the users to further improve SpaghettiLens.

6.1 Retrospect

The strict modular design of SpaghettiLens did pay off so far. It involves more work at the beginning, but offers much more flexibility for development of such a large and complex project. It helps, or even forces the developer to have a clear overview about which part has what tasks. You have to define APIs and protocols, preferably in advance, but this is needs to be done anyways for a project of this size.

During early stages of development, a lot of new features were used, especially in the client application. This posed a problem later, due to poor browser support or bad performance. For example at first the blending of the background image was done entirely in SVG, using SVG native filters and blending algorithms. It turned out however those filters are not optimized by most of even recent browsers. If only a small part of the picture changed, a full re rendering was triggered, in opposite of only re rendering the affected area. That lead to a huge performance impact that made the UI almost unusable. It is suggested that you use new features as conservatively and as little a possible. If you use an experimental feature, you should run exhaustive tests first.

The modularization of the application would allow to apply the development concept of unit tests. This was not applied in this project, but it would have been a good strategy with increasing size and module count. I would recommend using unit tests for a project of this size. I think that the overhead of writing unit tests will be recovered easily when implementing new features or changing existing modules.

6.2 Outlook

Since the project is rather huge (almost 50’000 lines of code) and already in operation (more than 9700 models for more than 850 different lenses created), quite a few bugs show up from time to time. But quite a few feature requests are coming in.

The most pressing new feature to be fully implemented is the ability to get multiple background images and merge them. While the basic program structures all already present, the interface to SpaceWarps does not yet allow to get the single band images.

With the “collaborative modeling” currently running, I see a demand for more advanced tools to actually manage models produced. In fact, it would be great to have an interface in order for scientists to easily upload one or a set of models and create a challenge that keeps track and visualizes all results and their relationship. This would also help modelers to get an overview what has already been done, and to continue the work on branch of the tree of the created models.

Scientists certainly would love to be able to compute more data and figures for selected models. This will be implemented while the next overhaul of the task system.

The client side event system for the input pane is a bit of a mess and could require a clean up, as does the API definition. This should be done at some time, but it is not important, since everything is actually working fine so far.

6.3 Concluding Remarks

I would like to thank Dr. Prasenjit Saha (University of Zurich) for offering me this exciting Master thesis and for all the support and motivation and Dr. Jonathan Coles (Université de Versailles Saint-Quentin-en-Yvelines) for providing the simulation backend GLASS.

Further I'd like to thank the other scientists involved with SpaceWarps, Dr. Phil Marshall (SLAC, Stanford University), PhD Anupreeta More (Kavli IPMU, University of Tokyo) and Dr. Aprajita Verma (University of Oxford).

Last but not least I thank all the SpaceWarps moderators and volunteers that used and tested SpaghettiLens with great enthusiasm and a lot of patience. Especially Elisabeth Baeten, Claude Cornen, Christine Macmillan, Jonas Odermatt and Julianne Wilcox for participating at the modeling challenge described in the yet to be released paper.

References

[1] Apple Inc. *OS X Human Interface Guidelines*. July 2012. URL: https://developer.apple.com/library/mac/documentation/userexperience/conceptual/applehiguidelines/Intro/Intro.html.

[2] Bert Bos. *CSS Specifications*. Aug. 2013. URL: http://www.w3.org/Style/CSS/current-work.

[3] Bert Bos et al. *Cascading Style Sheets, level 2 revision 1 – CSS 2.1 Specification*. Aug. 2002. URL: http://www.w3.org/TR/2002/WD-CSS21-20020802/.

[4] builtWith.com. *jQuery Usage Statistics*. Sept. 2013. URL: http://trends.builtwith.com/javascript/jQuery.

[5] Jonathan Coles. "Gravitational Lens Recovery with Glass: How to measure the mass profile and shape of a lens". to be published.

[6] Alexis Deveria. *Can I use...* Sept. 2013. URL: http://caniuse.com/#cats=CSS,PNG,DOM,CSS3,CSS2,Canvas.

[7] ECMA International. *ECMA-262 – ECMAScript Language Specification*. June 2011. URL: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[8] Roy T. Fielding and Richard N. Taylor. "Principled design of the modern Web architecture". In: *ACM Trans. Internet Technol.* 2.2 (May 2002), pp. 115–150. ISSN: 1533-5399. DOI: 10.1145/514183.514185. URL: http://doi.acm.org/10.1145/514183.514185.

[9] Ian Hickson and David Hyatt. *HTML 5 – A vocabulary and associated APIs for HTML and XHTML*. Jan. 2008. URL: http://www.w3.org/TR/2008/WD-html5-20080122/.

[10] ISO/IEC. *ECMAScript language specification*. June 2011. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/c055755_ISO_IEC_16262_2011(E).zip.

[11] ISO/IEC. *Information technology – Document description and processing languages – HyperText Markup Language (HTML)*. July 2012. URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=27688.

[12] Microsoft. *DevCenter – Design Guidelines*. Sept. 2013. URL: http://msdn.microsoft.com/en-us/library/windows/desktop/aa511436.aspx.

[13] P. Saha and L. L. R. Williams. "A Portable Modeler of Lensed Quasars". In: *aj* 127 (May 2004), pp. 2604–2616. DOI: 10.1086/383544. eprint: arXiv:astro-ph/0402135.

[14] solid IT. *DB-Engines Ranking*. Sept. 2013. URL: http://db-engines.com/en/ranking.