Freie Universität Berlin

Fritz Lekschas

SEMANTIC BODY BROWSER

A web-based tool for graphically exploring an organism’s body in respect to the CELDA ontology

Bachelor Thesis

Bioinformatics

Department of Computer Science and Mathematics

Freie Universität Berlin

The bachelor thesis has been realized at the Stem Cell Research & Knowledge Management group of Prof. Dr. Kurtz at the Berlin-Brandenburg Center for Regenerative Therapies, Charité Universitätsmedizin Berlin.

Supervisor and reviewer: Prof. Dr. Andreas Kurtz
Reviewer: Prof. Dr. Robert Tolksdorf

Submitted: 16th October 2012

Abstract

CELDA (Cell: Expression, Localization, Development, Anatomy) is an ontology, which comprehensively represents cells in the whole body. It serves as the main data source for CellFinder, a web-based data repository on cell types. By linking information from more than 15 other ontologies its structure gets complex and finding the desired data as easy as possible is vital. While a text-based search works best when the target is known by name, it fails when the user lacks this knowledge. In the latter case a tool is needed that requires only little or common knowledge.

The Semantic Body Browsers aims to provide an intuitive graphical way for exploring an organism’s body by means of annotated vector graphics. A simple mouse click highlights the target area and makes it possible to retrieve related information from CELDA or browse further along the three dimensions: resolution, development and species. The Semantic Body Browser is part of the CellFinder project and will is thereby integrated into the principal application.

The tool is implemented in JavaScript, HTML and CSS together with an API, written in PHP, and a MySQL database. The standalone version is available at http://sbb.cellfinder.org and the integrated tool can be found at http://cellfinder.org/browse.

Acknowledgement

First of all, I would like to express my deep gratefulness to Prof. Dr. Kurtz for giving me the ability to write my bachelor thesis at his research group and supervising me. I would also like to thank Prof. Dr. Tolksdorf for reviewing my thesis.

Furthermore, I am grateful for the support of the whole research group as well as the CellFinder team. I would like to give special thanks to Dr. Stachelscheid for the many valuable discussions and advises. I am also pleased to give thanks to Ms. Seltmann, who greatly helped me with the ontology.

Additionally, I am deeply grateful for my family who supported me in every way they could.

This thesis would not have been possible without the substantial help of all those people. Thank you!

Fritz Lekschas, Oct. 2012

CONTENTS

1 Introduction

1.1 Thesis

The bachelor thesis goal is the development of a web-based tool for graphically browsing the anatomy of the body of different species. The tool is part of the CellFinder project[1] and therefore maps units - regions of interest in an illustration - to the novel Cell: Expression, Localization, Development, Anatomy (CELDA) ontology which is the main data source of CellFinder.

The tool is implemented as a stand-alone web application[2] for the sake of the thesis first and integrated into the CellFinder platform[3]. As a proof of principle the graphical browsing is implemented for the human and murine kidney but includes the final set of features.

1.2 Motivation

CellFinder is a web-based data repository on cell types. It aims at being capable of handling, processing and integrating multi domain data. CELDA acts as the data structure, integrating eight different ontologies and eleven bridges while extending them. This leads to a huge and complex data source. CellFinder as a platform for cell types aims at researchers in various fields of biology and medicine and therefore it needs to be ensured that the information can be easily found and retrieved by users.

One way is the classical text based search, which works fine when the user knows exactly by name what he is looking for. Unfortunately, the approach fails when the user lacks this knowledge. Thus a tool is needed which requires only common knowledge about the domain to browse the data repository. The idea is to provide a graphical tool to browse an organism's body and retrieve insight information along the way. People interested or working in the field of research on

[1] http://cellfinder.org
[2] http://sbb.cellfinder.org
[3] http://cellfinder.org/browse

cells are familiar with the basic anatomy of the organs they are working with. Additionally, a tool that generally works like any common geographical map application[4] permits the user to explore new things quickly and intuitively.

As one of the working group's focus is on research in nephrology, the organ of choice is the kidney. The direct connection to the nephrology department of the Charité to the working group, led to the conclusion to primary concentrate on human. Apart from that, most of the current research concerning the development of the kidney has been realized in mouse. Additionally, CELDA integrates the Mouse Adult Gross Anatomy (MA) ontology, which holds comprehensive data on the mouse anatomy and development. Therefore, it has been decided to integrate mouse as the second species.

1.3 Scope

The Semantic Body Browser within the scope of the bachelor thesis is about the implementation of a web application for graphically browsing an organism's body in respect to CELDA. First of all this involved the development of a client-side web application written in JavaScript. Second, a server-side Application Programming Interface (API) is needed to handle the communication with a database and the CELDA ontology. Furthermore, illustrations had to be annotated and mapped to CELDA and a parser for automatic converting of Scalable Vector Graphics (SVG) to a JavaScript compatible file format had to be implemented.

[4] http://en.wikipedia.org/wiki/Comparison_of_web_map_services

2 Background Information

The Semantic Body Browser is part of the CellFinder project and should enhance the platform by providing an intuitive, graphical way of browsing CELDA. The following passages give an overview about the CellFinder project and the CELDA ontology as well as the anatomy and development of the kidney. Furthermore, technical background information in respect to the application is given.

2.1 CellFinder – A Stem Cell Repository

The CellFinder project aims at providing a comprehensive data repository on cell types [CellFinder, 2012]. The platform shall hold all types of data, which represents biological characteristics of cells grown *in vitro* as well as *in vivo*. The goal is to establish long-term storage and interoperability. In addition, CellFinder will serve tools to analyse and interact with cell related data.

To achieve these goals the CellFinder project can be divided into the four main work packages: text mining, protein and gene expression analysis, ontology and web application. A closer look is provided on the latter two fields as the Semantic Body Browser directly interacts with the ontology and is integrated into the web application.

2.1.1 CELDA Ontology

The main data structure behind CellFinder is the Cell: Expression, Localization, Development, Anatomy (CELDA) ontology. To successfully assimilate diverse types of data related to cells, CELDA integrates and extends eight existing ontologies as well as eleven bridges [Werner, 2012].

CELDA utilizes the BioTop ontology as a top-level structure and bridge to map domain-specific ontologies. To allow fast querying, the information of CELDA is additionally stored in a database and made available via an API. This approach permits to easily extent cell types with further data like gene or protein expression values or pictures that are not part of the ontology itself. Besides that, an API facilitates the development of tools for CellFinder using CELDA as a resource of information.

2.1.2 CellFinder – The Web Application

The main entry point for retrieving and interacting with data on cell types is the web application. This is where the user can search for anything related to cells, e.g. organs, tissues, developmental processes, genes and cells themselves. The browsing section is an alternative way to retrieve and display the data. A developmental tree for cell types is currently provided but should be extended by the Semantic Body Browser. Analysis tools allow the comparison of cells based on their gene and protein expressions. It is furthermore planned to permit the user to edit annotated data that is derived from text mining.

The application itself is written in PHP. The ontology and its relating data are queried through an API whereas everything else is directly retrieved from a MySQL database.

2.2 The Kidney

The kidney is an essential organ for the blood filtration. It is commonly ascribed to the urinary system because of its ability to filter waste, which is diverted to the urinary bladder. Furthermore the kidney plays an important role in maintaining the bloods homeostasis by regulating electrolytes, the acid-base balance as well as the blood pressure, which depends on the water-salt equilibrium [Wikipedia, 2012].

The next two passages describe the kidney's anatomy and the development of the nephron in detail.

2.2.1 The Anatomy

The two kidneys of mouse and human share a high anatomical similarity. They are bean shaped and have a concave surface called the renal hilum (Figure 1: 1), which connects the kidneys with the rest of the body. The renal artery (Figure 1: 2) and vein (Figure 1: 3) as well as the ureter (Figure 1: 4) enter the organ at this point. All three structures split up into several branches as they enter the kidney. The ureter for example migrates to the renal pelvis (Figure 1: 5), the main collecting tube that arises from merging the major calyces (Figure 1: 6). The major calyx itself originates from the minor calyx (Figure 1: 7). The kidney's parenchyma is the set of approximately seven to nine repetitive shapes, called the renal lobes (Figure 1: 8)

[Manski, 2012]. Each renal lobe consists of the outer renal cortex (Figure 1: 9) and a portion of the inner medulla called the renal pyramids (Figure 1: 10). The renal lobes in turn are the sum of a repetitive structure called nephron (Figure 1: 11).

The nephron is the functional unit of the kidneys. It regulates the bloods volume, pressure and electrolytes and moreover eliminates its wastes. Each nephron is composed of a blood filtering complex, the renal corpuscle (Figure 1: 12), and a tubular system (Figure 1: 13) in which the resorption and secretion takes place. The renal corpuscle consists of the glomerulus (Figure 1: 14) - a network of capillaries - and the glomerular capsule that is also known as Bowman's capsule (Figure 1: 15). The most important and prominent cell type involved in the blood filtering is the podocyte. The podocytes entwine the glomerulus and theirs numerous foot processes (Figure 1: 16) form slit diaphragms (Figure 1: 17) between each other, which perform the blood filtration. The renal capsule encloses the glomerulus and marks the start of the tubular system. This system is formed of several segments starting with the proximal tubule (Figure 1: 18), which is followed by the thin loop of Henle (Figure 1: 19), the distal tubule (Figure 1: 20) and finally the collecting duct (Figure 1: 21). The resorption and secretion of molecules mainly occurs at peritubular capillaries (Figure 1: 22) twining around the tubules.

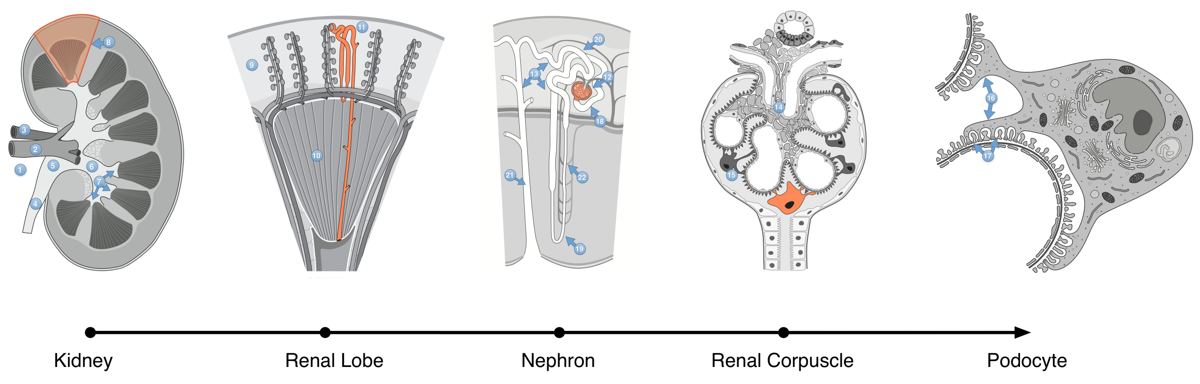


Figure 1: Anatomy of the kidney in human and mouse
Resolution increases in direction of the arrow. Each new level is highlighted in orange in the previous illustration.

2.2.2 Development of the Nephron

The development of the kidney proceeds through three stages: the pronephros, mesonephros and metanephros. Two distinct tissues, the ureteric bud - derived from the mesonephros - and the metanephric mesenchyme finally develop into the mature kidney [Pavenstädt, 2003][Davidson, 2009].

The pretubular aggregation of metanephric mesenchymal stem cells at the cap mesenchyme marks the beginning of a nephron's development. The first developmental stage - the renal vesicle - is a clustering of metanephric mesenchymal stem cells that develop into epithelial cells at the ureteric bud. The epithelial cells organize a body, which looks like a bean and is often referred to the comma-shaped body. This body already features a lumen, which will later make up the Bowman's space. As this body keeps growing its shape transforms into a more tubular system. The second developmental stage, called s-shaped body, distinguishes in two ways. First, the s-shaped body clearly shows the development of the tubular system and second, the proximal cleft is being invaded by angioblasts and precursor mesangial cells. While the s-shaped body keeps growing and the angioblasts and precursor mesangial cells mature, a first network of blood vessels can be seen. Furthermore does the lower limb transform into the glomerular capsule and an immature renal corpuscle can be identified. At this point the structure is recognized as the capillary loop nephron, the third developmental stage of the nephron. The final step invokes further maturing and elongation of the tubular system, which connects with the collecting duct system.

The developmental processes begin with the 20th Theiler stage in mouse [Harding, 2011] [McMahon, 2008], respectively 17th Carnegie stage in general according to Haudry et al. (Figure 2) [Haudry, 2008]. The pretubular aggregate, renal vesicle and comma-shaped body develop within approximately one day. The s-shaped body can be seen from the 21st Theiler / 18th Carnegie stage on and develops to the capillary loop nephron around the 22nd Theiler / 19th Carnegie stage. As a normal mature kidney consists of roughly 800.000 – 1.5 million nephrons [Guyton, 2006], the development does not start simultaneously and thus the whole process continues until the 23rd Carnegie stage.

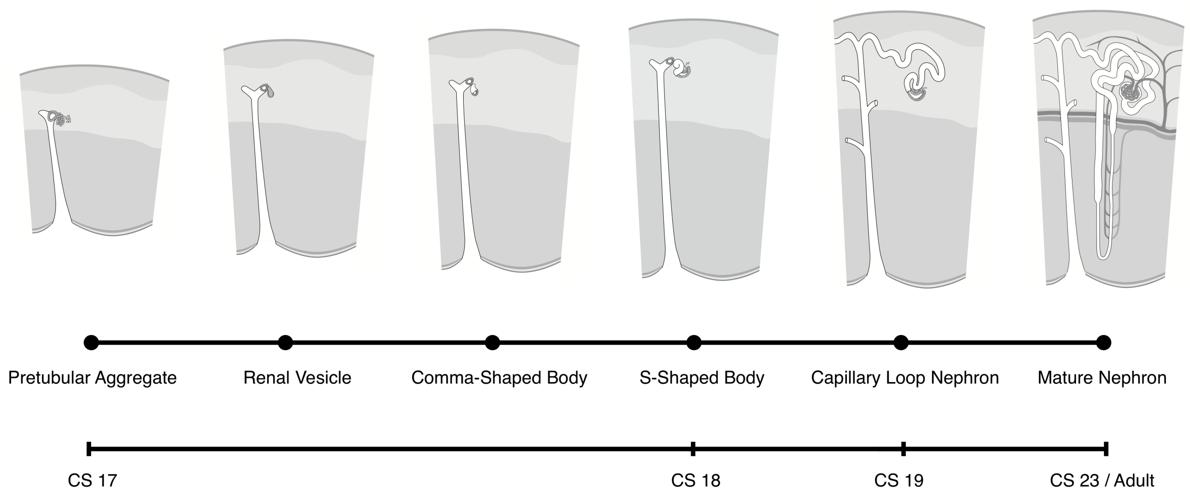


Figure 2: Developmental stages of the nephron in human and mouse
The timeline does only highlight the earliest beginnings of a developmental stage as the process is dynamic and the timeframe is rather flexible. The start and end point are fixed, though.

2.3 Vector Graphics

Vector graphics are a type of computer graphics that exactly describe objects with the use of basic geometric forms like a line, circle, rectangle or path. This differs from raster graphics, which are based on a grid of colours called pixels. The biggest advantage of vector graphics is the lossless scalability and with the establishment of the XML based Scalable Vector Graphics (SVG) it is even possible to directly manipulate vector graphics on the web.

2.3.1 Vector vs. Raster Graphics

Vector graphics, as the name indicates, are made up of vectors. These vectors define lines and curves at certain locations and thus describe an object’s geometrical characteristics mathematically. Besides the outlines of a shape, other information can be defined such as the filling colour or a stroke. As a consequence, vector graphics need to be rendered before they can be displayed but the rendering process is resolution independent (Figure 3). Thus vector graphics are mostly used for fonts, glyphs and abstracted illustrations. On the other hand raster graphics do not describe any objects, they only assign a location and a colour to each pixel. As they represent the nature of computer displays, which are raster devices, they do produce the same visual representation on any machine. Apart from that they are preferable over vector graphics when it is needed to display a large amount of details like in photographies. Vector graphics are not capable of describing such a high number of information efficiently and thus would dramatically increase in their file size and look unrealistic.

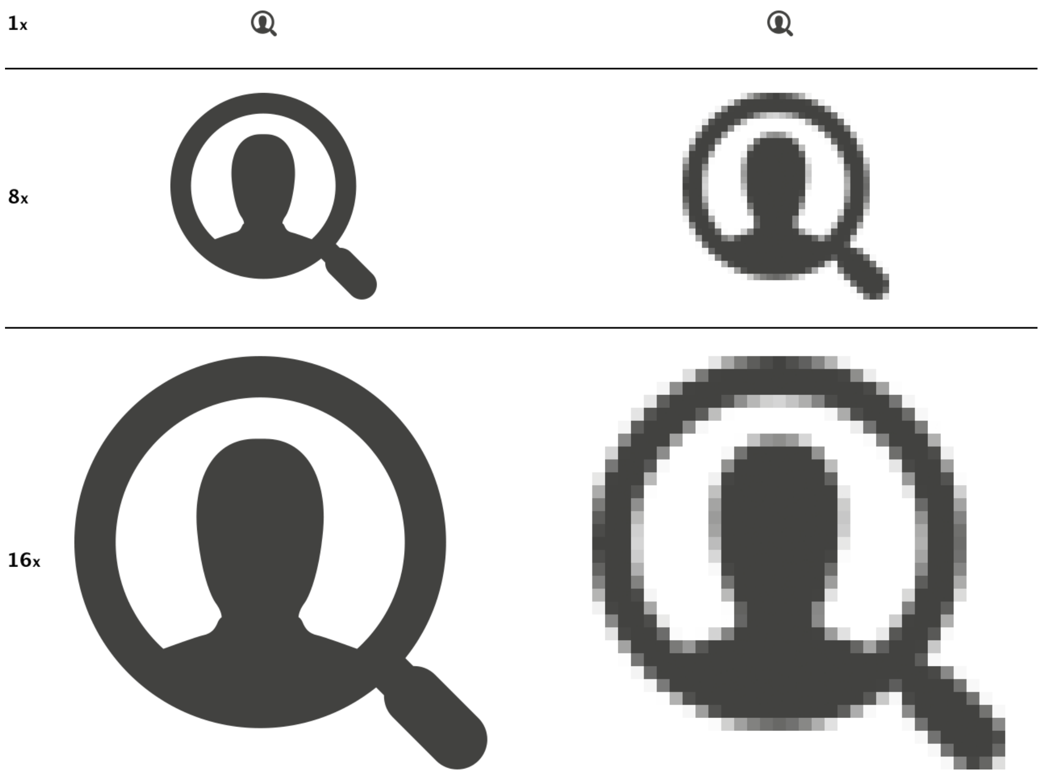


Figure 3: Vector vs. raster graphics
The vector graphic (left side) remains sharp at any resolution whereas the raster graphic (right side) cannot be magnified without loosing clarity.

2.3.2 Scalable Vector Graphics

Scalable Vector Graphics (SVG) is a specification for two-dimensional vector graphics defined by the World Wide Web Consortium (W3C) [W3C, 2011]. It is based on the Extensible Markup Language (XML) and can be used in all modern web browsers, even though it should be noted that none of the current browsers support all features [Schiller, 2011]. SVG are plain text files and can thus be edited with any text editor. This provides the ability, amongst others, to script and compress the image. SVG follows a strict structure due to the fact that it is based on XML (Listing 1).

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <svg xmlns="http://www.w3.org/2000/svg"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:ev="http://www.w3.org/2001/xml-events"
        version="1.1" baseProfile="full"
        width="700px" height="400px" viewBox="0 0 700 400">
3.      <rect x="100" y="100" width="500" height="200" fill="#fff" stroke="#000" stroke-width="20"/>
4.      <circle cx="300" cy="80" r="50" stroke="#000" fill="#0f0" stroke-width="2"/>
5.      <ellipse cx="0" cy="0" rx="100" ry="50" fill="#ff0" />
6.      <path d="M150 0 L75 200 L225 200 Z" fill="#00f" stroke="#000" stroke-width="5" />
7. </svg>
```

Listing 1: SVG example

The first line defines which XML version and text encoding is used. The second line sets a container with its dimensions as well as the view box. Lines three to six are examples, illustrating the basic geometric shapes SVG is capable of. These four elements also represent all relevant elements in terms of the Semantic Body Browser.

2.4 A JavaScript Web Application

JavaScript (JS) has been invented by Netscape to compensate the static nature of HTML. Its goal was to make websites more interactive and has long been used to solely enhance the feeling of a website's functionalities. The rise of Asynchronous JavaScript and XML (AJAX) led to the development of several libraries, which streamline the behaviour in different browsers. This opened up a way to use JavaScript for more complex tasks. With less effort it was possible to make a website behave more like a desktop application.

A JavaScript web application combines several technologies apart from JavaScript itself. Often, HTML in association with CSS shapes the look. An API is usually used to enable the communication between JavaScript on the client-side and a database on the server-side. All aspects as well as essential frameworks, libraries and tools are described in the following passages, beginning with the language JavaScript.

2.4.1 JavaScript

JavaScript is an object oriented dynamic scripting language. It is often used in the Web but can also be found in non-browser environments[5].

JavaScript has been invented by Brendan Eich, an engineer at Netscape, in 1995 and first released with the Netscape 2. It was former called LiveScript but in an attempt to benefit from the popular Java language, it had been renamed to JavaScript. It should be noted that the word “Java” is the strongest similarity between the two languages. Based on the request of Netscape, JavaScript has been standardized by the ECMA International and released in 1997 under the new name ECMAScript. At the point of this writing all modern web browsers support version 5.1 of ECMAScript. Thus, JavaScript can be seen as an implementation by Mozilla of the current ECMAScript standard. But for the sake of simplicity, the term “JavaScript” will be used to represent any implementation of ECMAScript.

JavaScript's first purpose was to enrich websites by providing ways for dynamic interactions. By standardization of JavaScript it is no longer fixed to the browser and can theoretically be used in any environment. This still causes confusion and is one of the main reasons why JavaScript has been entitled as one of the most misunderstood programming languages [Crockford, 2001].

JavaScript is a prototype-based programming language with first-class functions [Mozilla 2012]. It can be used with an object-oriented, imperative or functional design paradigm. Differences between other object-oriented programming languages are that JavaScript has no classes and uses prototypal inheritance for reuse of code. It is furthermore worth to note that functions themselves are only objects, which leads to the ability to make use of first-class functions.

2.4.2 Document Object Model

Working with JavaScript that runs in a browser environment implies working with the Document Object Model (DOM). HTML and XML are markup languages that follow a strictly hierarchical structure. Manipulation of the structure requires an interface that

[5] http://en.wikipedia.org/wiki/JavaScript#Uses_outside_web_pages

defines how to access and traverse elements: the Document Object Model (Figure 4).

The W3C has standardized the DOM in 1998. The current Version is 3, released in 2004 [W3C, 2012]. But still the implementations of the DOM among various rendering engines show significant differences [Resig, 2011].

```
<html>
    <head>...</head>
    <body>
        <header>
            <h1 id="logo">...</h1>
        </header>
        <section class="text">
            <h2>...</h2>
            <p>...</p>
            <p>...</p>
```

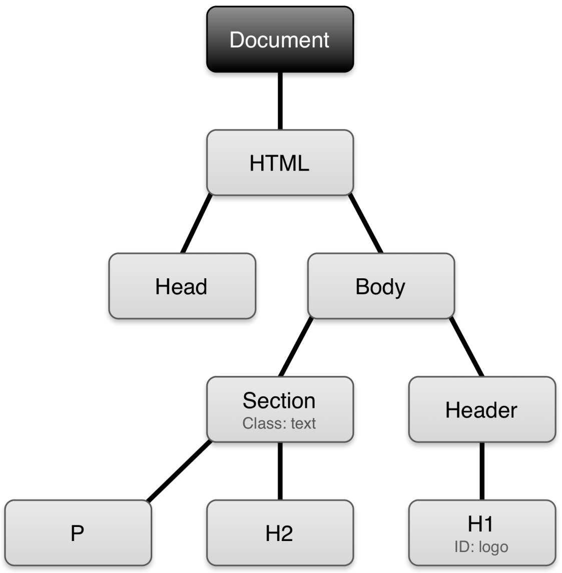


Figure 4: HTML and DOM
On the left is an HTML document and on the right the related DOM tree. The dark "document" node is the root and does not represent any HTML element. The DOM specification moreover describes a set of methods for querying, traversing and manipulating the DOM.

2.4.3 AJAX

AJAX - Asynchronous JavaScript and XML - describes a concept for asynchronous communication between the browser and a server. One technique behind, which is called XMLHttpRequest, accomplishes to execute HTTP requests on the client-side without having to reload the actual page. This means that content from another resource can seamlessly be loaded behind the scenes and changed on the fly without interfering the display of the current page.

The rise of AJAX began in 2005 [Gerett, 2005] but it wasn't a novel concept. It is more a collection of several technologies combined under a new name (Figure 5). Even though the technologies exist since 1998, their diverse implementation in different browsers holds back the adaption. This changed with the emergence of several frameworks and libraries that streamline the usage of asynchronous server requests and DOM manipulation. The most popular library nowadays is jQuery [W3Techs, 2012].

It should be noted that JavaScript technically cannot interact with any files or databases directly. Talking about manipulations solely means manipulations of the current DOM. DOM manipulation are of temporal nature, thus they are lost after a page reload. This can only be avoided by calling other technologies via the XMLHttpRequest.

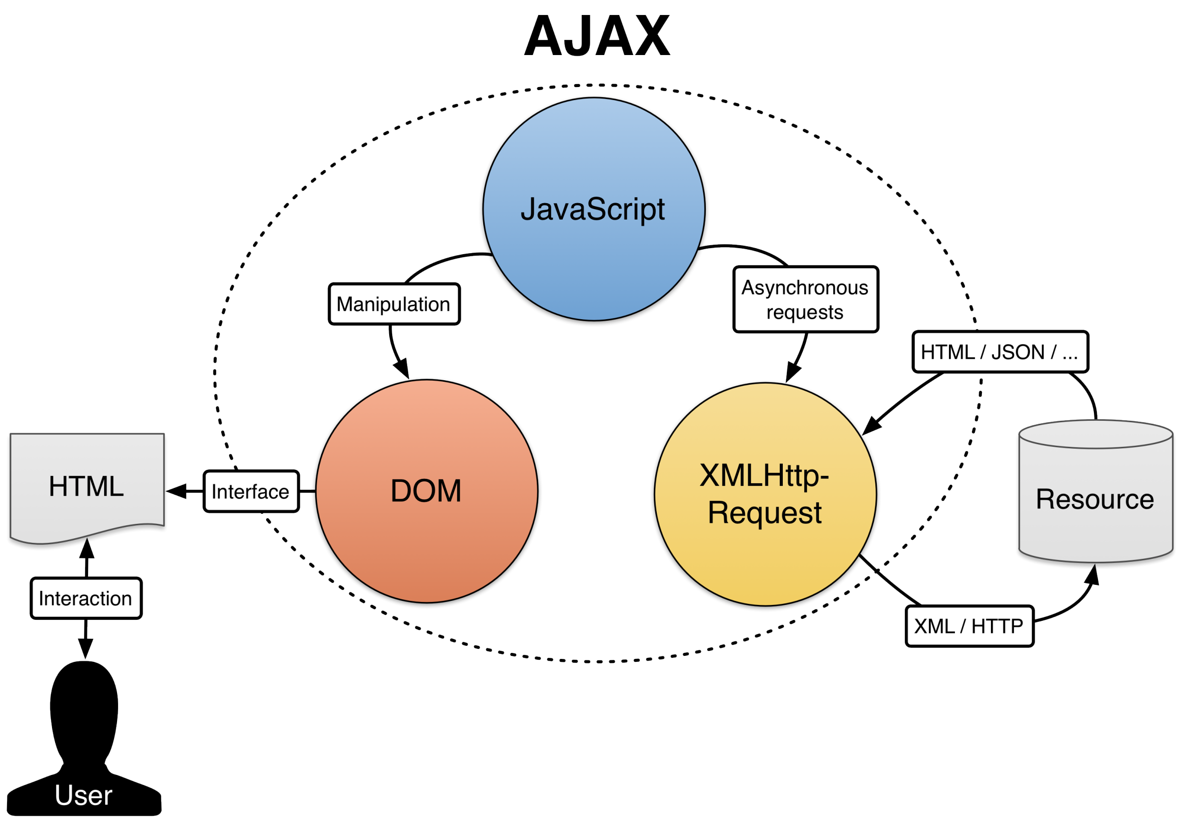


Figure 5: AJAX
JavaScript is the key of AJAX. With it, one can asynchronously manipulate the DOM and call XMLHttpRequests. All technologies together enhance the interaction with a website by avoiding page reloads.

2.4.4 Frameworks, Libraries & Tools

JavaScript is a powerful language and has proven to be capable of more than just an enhancement for HTML. However, writing application in plain JavaScript comes with a huge drawback. As discussed in the previous chapter, AJAX technologies in their nowadays use are not fully implemented in respect to the standardization and vary among different engines in many ways. This causes the need for so-called boilerplate code to streamline the behaviour and output of functions. By relying on a number of well developed frameworks and libraries this repetitive work can be saved.

The Semantic Body Browser relies on one framework and three libraries which all specialized on serving different features. The frameworks and libraries themselves and their purpose are introduced in the following passages.

2.4.4.1 AngularJS

AngularJS[6] is an open-source JavaScript framework. It implements the Model-View-Controller (MVC) design pattern, which helps to keep code clean. Additionally it extends HTML to make it capable of dynamic content.

The MVC design pattern describes a programming paradigm, which separates code into three parts: models, views and controllers (Figure 6). A clear structure and the separation of different sections make it easier to maintain, extend and debug code. The model hereby describes the code that interacts with and stores the data. A view is the representation of the data on the screen. Finally the controller is responsible for the application’s logic; it processes inputs, delegates’ tasks and prepares the output.

[6] http://angularjs.org

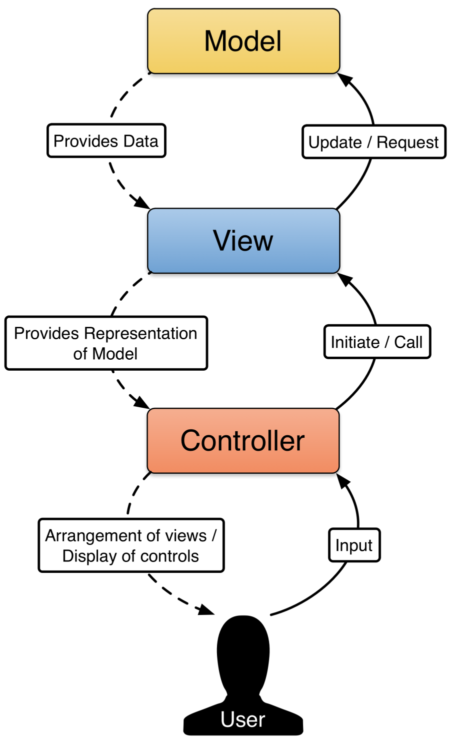

Model
Provides Data
Update / Request
View
Provides Representation of Model
Initiate / Call
Controller
Arrangement of views / Display of controls
Input
User

Figure 6: Original MVC design pattern

The original Model-View-Controller pattern describes by T. Reenskaug in 1979 [Reenskaug, 1979].

In AngularJS the model is any JavaScript object that holds some kind of data. The view is normal HTML and CSS, extended with AngularJS specific attributes and elements. The controller is represented as a JavaScript function.
AngularJS separates code even further in directives, services and filters, to ease the maintainability. Modern JavaScript applications make significant use of AJAX and thus invoke a number of asynchronous tasks like DOM manipulations and XMLHttpRequests. These can be difficult to test and may be used several times throughout the application. AngularJS therefore advises to separate all kinds of DOM transformation in directives and define services for common tasks. Filters are intended to work similar as UNIX filters, thus they should deal with data transformation.

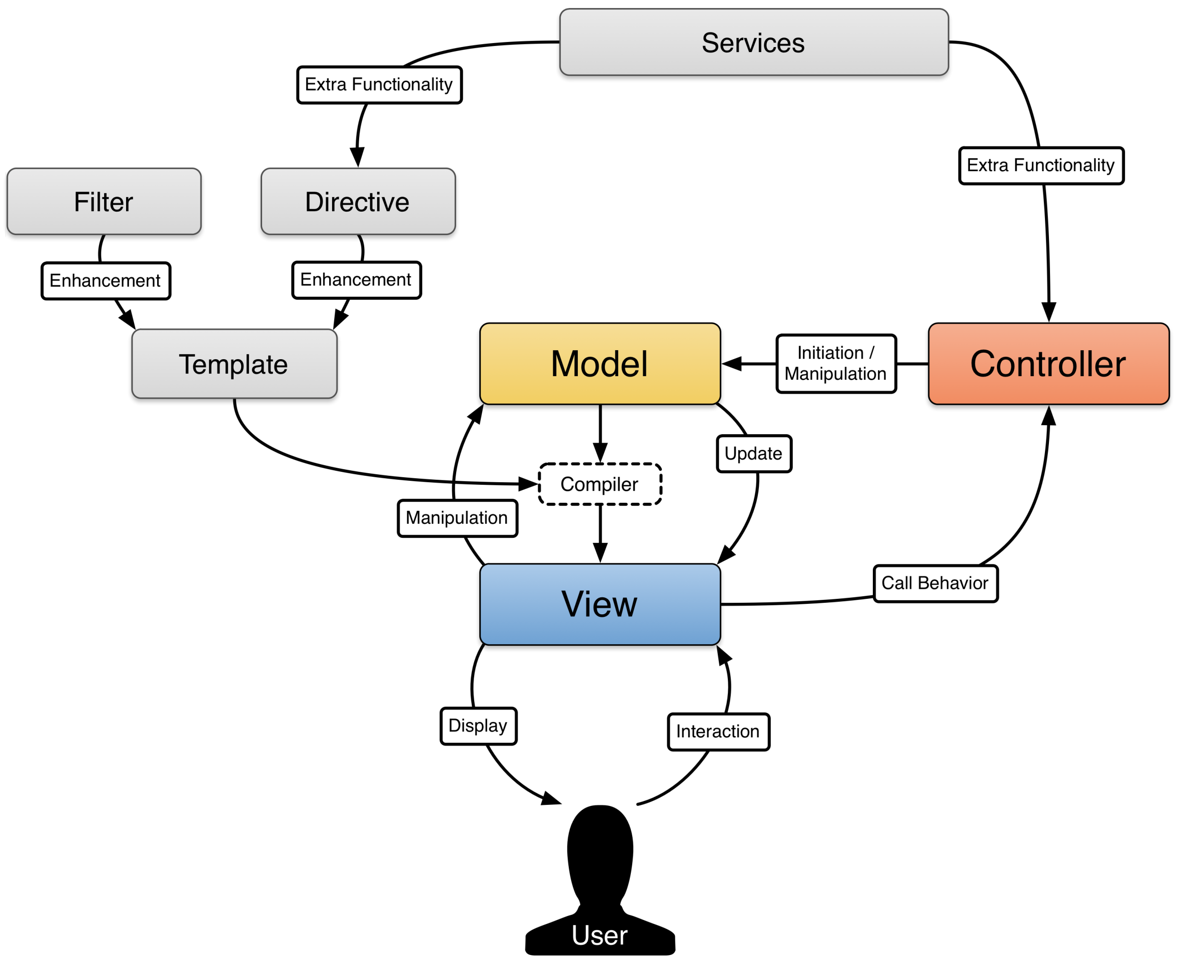


Figure 7: MVC implementation of AngularJS
In AngularJS the user interacts with the view, which is composed of a template and the model. The view knows about the model and the controller and can communicate with both. Services can be injected into controllers and directives. Directives and filters are used to enhance the static nature of HTML.

One of the biggest advantages of AngularJS is hereby the elimination of boilerplates. This results in much shorter and cleaner source code (Listing 2).

```
<!doctype html>
<html ng-app>
  <head>
    <script src="js/angular.min.js"></script>
    <script type="text/javascript">
      function Ctrl($scope) {
        $scope.names = ["Peter", "Mayer", "Dirk"]
        $scope.add = function () {
          if (this.name) this.names.push(this.name)
10.         }
11.       }
12.     </script>
13.   </head>
14.   <body>
15.     <div ng-controller="Ctrl">
16.       <form ng-submit="add()">
17.         <input ng-model="name"><input type="submit" value="Add">
18.       </form>
19.       <ul>
20.         <li ng-repeat="name in names">{{ name }}</li>
21.       </ul>
22.     </div>
23.   </body>
24. </html>
```

Listing 2: AngularJS: example application

At first the browser loads the HTML and parses the DOM. Then the AngularJS source code is loaded (line 4). After AngularJS is ready the application will be bootstrapped in the scope of the ng-app directive (line 2). This binds JavaScript to DOM elements.

Within the scope of the Ctrl controller – highlighted in orange – it is possible to access its models and functions. The models – highlighted in blue – can hereby be created before (line 7) or during the runtime (line 17). When a user enters a new name, the underlying model will automatically be updated. Submitting the form pushes the current value of the model on the array “names”. Because AngularJS uses two-way data binding, changes on any site, JavaScript or HTML, will be reflected immediately and thus the list of names (line 20) will be recompiled.

AngularJS defines any object that is reachable within a scope as a model. Therefore it can be predefined like “names” (line 7) or generated during the runtime like the two models names in line 17 and 20. The example thus has three models in two different scopes.

The controller handles the logic on a certain scope and is represented as a JavaScript function (lines 6 – 11). A controller thus defines the initial state and behaviour of a scope. In the example, the controller initialises “names” (line 7), which can be modified by the “add” method (lines 8 – 10). A controller should never manipulate the DOM directly which is separated into directives.

One of the build in directives is ng-repeat which loops over an array and renders HTML for every element. A directive usually has its own scope to avoid conflicts. The example demonstrates the existence of two different scopes by using the name “name” for the models twice. Even though both models have the same name they represent different data

as they belong to distinct scopes. The model in line 17 belongs to the Ctrl controller and the other model in line 20 is part of the ng-repeat directive.
The HTML, that follows after the application is initialized (line 2), reflects the template. AngularJS adds more functionality to the HTML by custom directives (ng-repeat line 20), attributes (line 15: “ng-controller”) and expression. The example illustrates one expression: “{{ name }}” (line 20). These expressions can be interpreted as JavaScript expression with limitation in their feature set. In this case the value of each name in the list of names is bond to the view.

2.4.4.1.1 The AngularJS Seed

AngularJS provides a template web application which serve as a blueprint when starting a new project. The AngularJS Seed[7] provides a basic skeleton, which invokes the standard AngularJS framework files and wires them together. This gives an instant start for the development and a clear structure to follow (Table 1).

Method			Description
./app			Application root
	./css		Contains all Cascading Style Sheets
	./img		Contains images
	./index.html		Main template
	./js		Contains JS application files
		../app.js	Contains general configuration of the application
		../controllers.js	Contains the applications controller
		../directives.js	Contains the applications directives
		../filters.js	Contains the applications filters
		../services.js	Contains the applications services
	./lib		Contains JS libraries
		./angular	Contains AngularJS specific libraries
	./partials		Contains templates

Table 1: AngularJS Seed: file structure

[7] https://github.com/angular/angular-seed

2.4.4.2 jQuery

DOM manipulation is an essential tool for every JavaScript application. The asynchronous character requires JavaScript to load content dynamically and then update the DOM to visually reflect the changes. jQuery[8] is an open-source JavaScript library that provides tools for easy, cross-browser manipulation and traversing the DOM as well as XMLHttpRequests. It works perfectly together with AngularJS as AngularJS already provides a small subset of jQuery.

2.4.4.3 Raphaël

Raphaël[9] is an open-source JavaScript library that aims at simplifying and streamlining the work with vector graphics on the web. It implements SVG and uses the Vector Markup Language (VML) as a fallback for Internet Explorers older than version 9. This means that the elements of the vector graphics are not only displayed but also registered as DOM elements, which make it possible to manipulate them with JavaScript.

2.4.4.4 Hammer.js

jQuery and AngularJS do only support mouse and keyboard events. Hammer.js[10] is a lightweight JavaScript library that implements a rich set of touch events. With the help of Hammer.js it is possible to enable the Semantic Body Browser for the usage on a number of touch devices, like smartphones and tablets.

2.4.4.5 Grunt

When working with JavaScript one comes across a number of repetitive tasks. This often includes concatenating, minifying and testing the code. Performing these tasks manually can take a lot of time. Grunt[11] is an open-source command line tool that can handle common tasks including the one named before.

---

[8] http://jquery.com
[9] http://raphaeljs.com
[10] http://eightmedia.github.com/hammer.js/
[11] http://gruntjs.com

Grunt is written in JavaScript and needs NodeJS[12] - a JavaScript server environment - to be installed. It is configured by a single file, called grunt.js or simply gruntfile, in which the tasks that should be performed are defined (Listing 3).

```
1.  module.exports = function(grunt) {
2.    grunt.initConfig({
3.      concat: {
4.        dist: {
5.          src: ['src/a.js', 'src/b.js'],
6.          dest: 'src/all.js'
7.        }
8.      },
9.      lint: {
10.       files: ['lib/*.js']
11.     },
12.     min: {
13.       dist: {
14.         src: ['src/all.js', 'lib/*.js'],
15.         dest: 'src/min.js'
16.       }
17.     },
18.     qunit: {…},
19.     server: {…},
20.     test: {…},
21.     watch: {…}
22.   });
23.   grunt.registerTask('default', 'lint min');
24. }
```

Listing 3: Gruntfile example

Grunt loads the module (line 1) and looks for the project configurations (lines 2 – 22). If no tasks are specified, the default tasks are performed (line 23). An object inside of the configuration defines a task, like “min” in line 12. Each task has a specific set of attributes that act as its configuration.

[12] http://nodejs.org

2.4.4.6 Slim

Slim[13] is a PHP framework for small to medium sized web applications and APIs. Its goal is to provide a comprehensive set of tools while keeping a small footprint. Besides other use cases, Slim makes it possible to set up a RESTful - Representational State Transfer - web service, respectively API. Such an API is stateless which means that every query is treated independently and is not influenced by former actions. This implies that a certain URL, following a defined pattern, represents all of a query's parameters (Figure 10).

[13] http://www.slimframework.com

3 Application Design

The Semantic Body Browser follows the general structure of a web application and separates the actual application from the data. The JavaScript application operates on the client side and communicates with a database via an API. Both, the API and the database are located on the server side (Figure 8).

The following passages describe both sides in detail and explain how they communicate and work together.

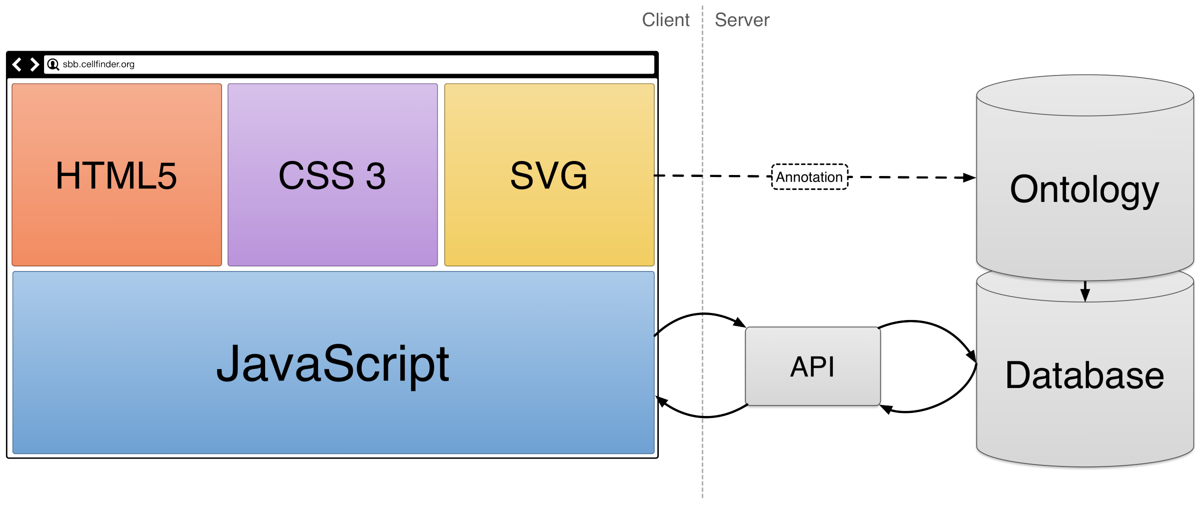


Figure 8: System architecture

3.1 Client-Side

Main part of the Semantic Body Browser is the JavaScript application, operating on the client-side. It processes the user inputs, loads the required illustrations and data and renders the views. The JavaScript application is build upon an AngularJS Seed, which handles all the business logic. Additionally, the three libraries: jQuery, Raphaël and Hammer.js are integrated. JQuery is mainly used for DOM manipulation. Raphaël adds functionalities for working with SVG, which is needed to display and interact with the illustrations. Hammer.js enhances the application with touch gestures to make it able to response on touch devices, like smartphones and tablets.

For the Semantic Body Browser only two controllers are needed. One handles the displaying and manipulation of the illustration and another one simply provides texts with background information. Because AngularJS separates the business logic and the DOM manipulation a couple of smaller directives are needed as well as a more complex one for dealing with SVG. Furthermore, services for retrieving the illustrations and querying the API are needed. Finally, two views display the text and one the illustrations.

3.2 Server-Side

To follow the best practice of separating code from content, a MySQL[14] database server is responsible for the data storage. Because JavaScript itself has no possibility to directly query the server an API, build with Slim, is used to establish the communication between the two parties.

3.2.1 Database

The database needs to be able to hold information about the three dimensions of browsing: resolution, development and species. This is achieved by assigning each view, which is representing an illustration, a certain level of resolution, a developmental stage and a species. Each view contains a number of units. A unit can be interpreted as an area of interest within an illustration, like an organ, a tissue or a cell, which the user can interact with. Furthermore, the database design represents definitions, synonyms and pictures (Figure 9).

A view is identified by a unique name, which can possibly be anything. It furthermore consists of a unique combination of a developmental stage, a species and a level of resolution. These values are linked as foreign keys from the related tables to ensure integrity. Each view is mapped to an ontological term via a Uniform Resource Identifier (URI). If the view's name differs from the related illustration's name, a custom image source can be specified. Finally, each unit can have a custom title and link to a certain prefix. The prefix tables serves as a helper to reduce the length of unit names and makes their sorting more useful. For example every non-unique region or cell within the kidney could be prefixed with "renal" even

[14] http://www.mysql.com

though this prefix does only make sense when the region or cell is referenced from another scope.

A developmental stage is composed of a name, a precursor stage as well as a successor stage. The latter two are self-referencing keys to other developmental stages from the same table and used to calculate the order. Species have an ID, which matches their colloquial name and their Latin counterpart. A name, as well as their parent describes levels of resolution.

A unit is identified by a name and the view in which it can be found. Therefore the “view” attribute act as a foreign key. Each unit is mapped to a certain term in CELDA and can have a custom title and prefix. The possibility to zoom into a unit is processed automatically by checking whether a view with the units name does exist, keeping in mind the species and developmental stage. Apart from that a custom zoom can be specified. This is especially useful when the next level of resolution consists of a set of units.

The picture, synonym and definition tables link additional data to a certain URI. Their structure is based on their counterparts in CellFinder to allow quick and easy migration.

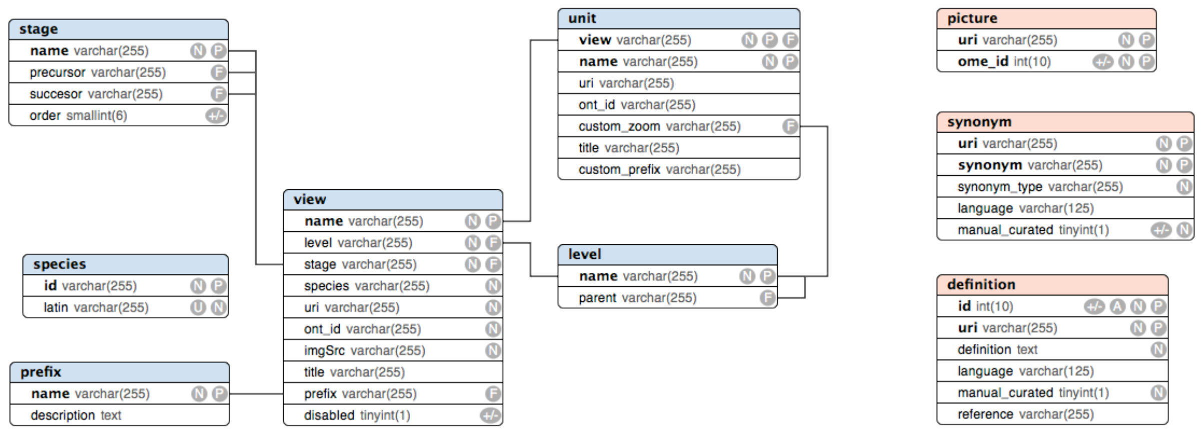


Figure 9: Database design

The tables highlighted in blue describe the browsing. The other three tables highlighted in orange hold supplementary information about the view and the units.

+/-	Unsigned number	N	Not null
A	Auto increment	P	Primary key
F	Foreign key	U	Unique

3.2.2 API

The API is a single file application that bridges queries of the database. Because the application only allows visual manipulation of certain styles of an illustration, there is no need for data manipulation of the database. Thus the API needs functions of retrieving data only. As the client-side application is written in JavaScript the response format of choice is JavaScript Object Notation (JSON).

The API needs to track a number of different Uniform Resource Locator (URL) patterns to retrieve information that is being stored in the database (Table 2). Furthermore does the API provide wrapper methods, which themselves call a set of other methods, to reduce the amount of HTTP requests needed to get the desired data.

Method	URL Pattern	Description
getAll	/<VIEW>	Wrapper method that queries and combines: getView, getUnits, getDefinition, getSynonyms, getPictures and getUnitData.
getView	/views/<NAME>	Gets data related to a view.
getUnits	/units/<VIEW>	Gets units for a view and checks, which are zoomable.
getUnitData	/data/<VIEW>	Gets definitions and synonyms for all units within a certain view.
getSynonyms	/synonyms/<URI>	Gets synonyms of a given URI.
getPictures	/pictures/<URI>	Gets pictures of a given URI.
getDefinitions	/definitions/<URI>	Gets definitions of a given URI.
findSimilarViews	/search/<LEVEL>	Searches for similar views at different developmental stages or species while holding the same level of resolution.
findZoomViews	/search/<STAGE>/<SPECIES>	Searches for views that can be zoomed in and parent views.
findView	/search/<LEVEL>/<STAGE>/<SPECIES>	Searches for a view with a certain name.

Table 2: API

The table lists the available methods to retrieve information from the database. Following the URL patterns, which are relative to the API's location, triggers the methods. Parameters – written in uppercase and encapsulated with angle brackets – are strings and mandatory.

4 Results

The Semantic Body Browser is a tool for graphically navigating through an organism's body and across species. In the scope of this thesis the exploration of the human and murine kidney has been implemented on the basis of sixteen illustrations. Starting at the bodies level, one can zoom all the way in to the podocyte. It is possible to switch the species at any time and to trace six developmental stages of the nephron (Figure 2). All illustrations are annotated and 395 interactive units are mapped to the CELDA ontology in total. Along the development of the application 41 new species-specific terms have been added to CELDA in order to be able to avoid the use of general terms (Table 3).

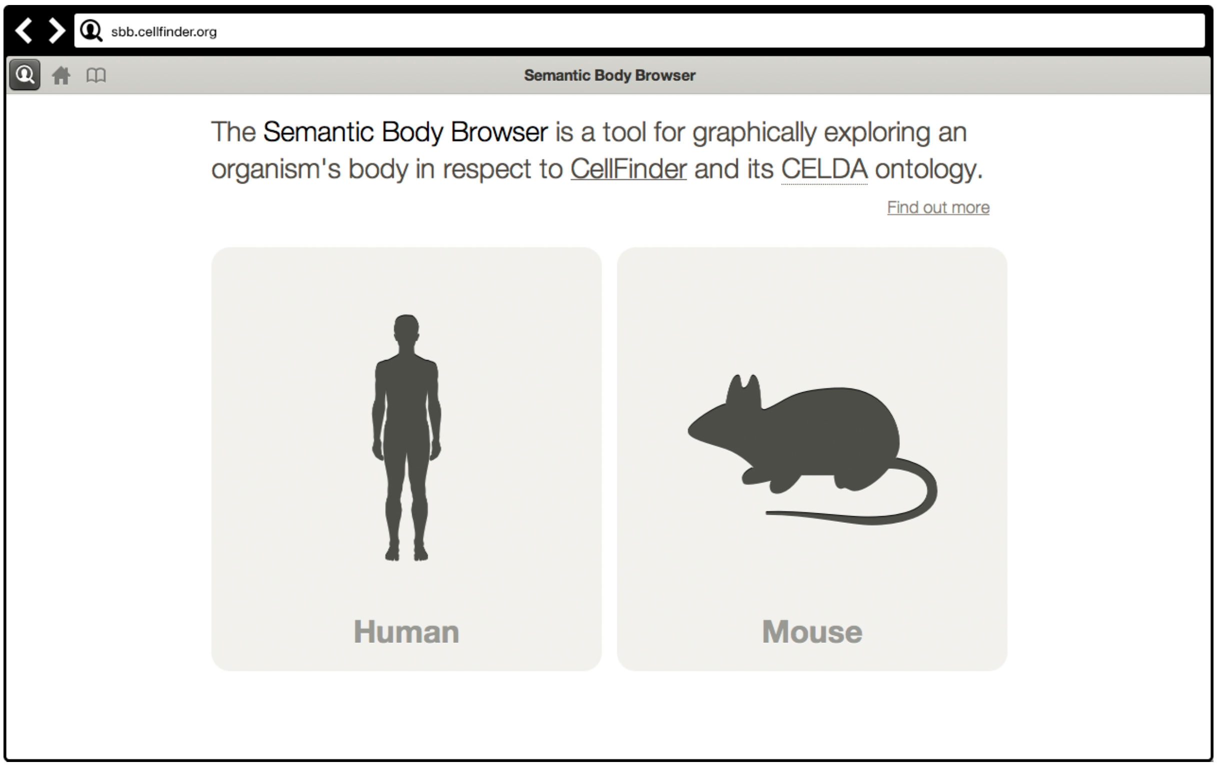
The following passages present the results and explain the implementation of the Semantic Body Browser in detail.

4.1 A Walkthrough

To better understand the applications features this walkthrough provides screenshots and explanations of what the user can do with the Semantic Body Browser.

Starting with the home screen (Figure 10) that is displayed when the website is visited for the first time, a user is confronted with a short description and the choice between the two species: human and mouse. To make it as straight forward as possible the two buttons for the species take up two third of the whole screen and are emphasized with pictograms for their respective species.

When the user clicks on the human or mouse button, the main view of the application is loaded. It is basically divided into three sections: the top bar, the area for the illustration and the sidebar on the right.



The Semantic Body Browser is a tool for graphically exploring an organism's body in respect to CellFinder and its CELDA ontology.

Figure 10: Semantic Body Browser: home screen

Top bar

The top bar has three main areas. On the left is the navigation for the application itself. You can get background information (Figure 11: 1), go back to the home screen (Figure 11: 2) or save and restore bookmarks (Figure 11: 3). In the middle of the top bar is the name of the current view displayed (Figure 11: 4). The area on the right holds navigations for the current view. If available it is possible to change the resolution (Figure 11: 5), meaning to zoom in or out, to trace developmental stages (Figure 11: 6) or to switch the species (Figure 11: 7). Furthermore the search input allows filtering the units of the current view (Figure 11: 8).

Side bar

Information concerning the view can be found in the right side bar. Most important, the units are listed here (Figure 11: 9). To easily keep track of all units they are grouped by their first letter, which is indicated on the right. It should be mentioned that common prefixes for organs are excluded during the grouping process but still displayed to make sure that the official name is present. Units that are marked with a small magnifier inform the user that a higher resolution of this unit is available. A single click on a unit highlights it (Figure 11: 10) and the related region in the illustration (Figure 11: 11). A double click on a unit zooms the user in, in case it is available.

Below the units is a section, which holds general information about the current view (Figure 11: 12). If available a short definition or description is given as well as common synonyms. This information is derived from the CELDA ontology. In this respect, the URI is given and links to the related site of the CellFinder application.

The last available section within this sidebar is the picture panel below (Figure 11: 13). It is minimized by default but can be opened with a click on its name. Microscope pictures are provided by Omero[15] - an image repository for managing and analysing microscope pictures. If more than one picture is available the user can flip through them by clicking on the small arrows. A click in the centre of the picture opens up the detailed image viewer of Omero that is integrated into the CellFinder application. It should be noted that this feature is currently only present for the renal corpuscle, because there are currently only very few microscope pictures available. But still it has been implemented as a proof of principle.

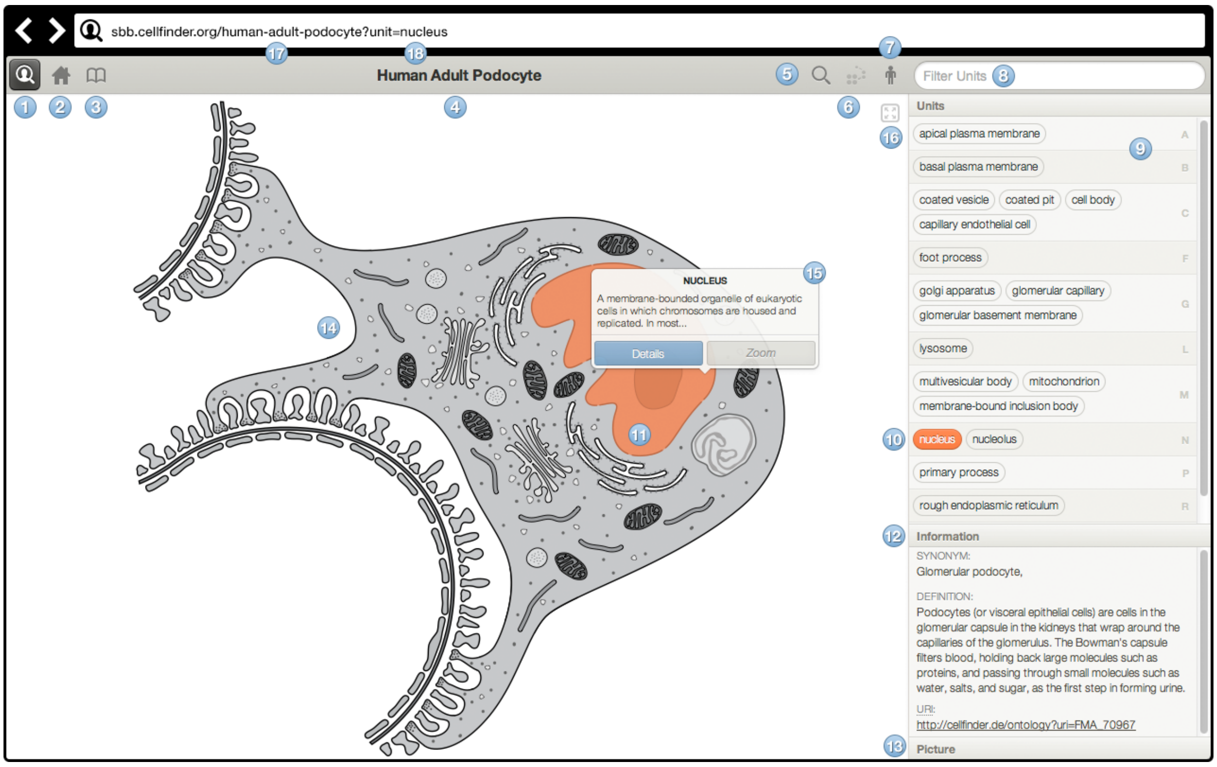
Illustration

The biggest part of the screen is displaying the illustration (Figure 11: 14). The user has the possibilities to drag the illustration by pressing the mouse and moving the cursor. By scrolling inside of the container the illustration gets magnified or shrunk to a certain extend. Most important, it is possible to interact with the units by simply clicking on them. The mouse click highlights the target and similar objects of the same type. Furthermore, a double click on any of the units opens a dialog (Figure 11: 15) that gives a short definition or description as well as the possibility to open that unit with CellFinder or to zoom into it, if available. Is the double click performed outside of any unit, the current highlighting will be reset. The illustration's initial position and scaling can be recovered by the button in the top right corner of the illustration panel (Figure 11: 16).

[15] http://www.openmicroscopy.org/site/products/omero

Deep Linking

Finally, to make referencing possible the application allows deep linking, which means linking to a specific state of browsing. Any changes on the screen are reflected in the URL, even if the back or forth button is used. The path followed by the root describes the view (Figure 11: 17 “human-adult-podocyte”) and the currently active unit is given by the search parameter “unit” (Figure 11: 18 “nucleus”).



NUCLEUS

A membrane-bounded organelle of eukaryotic cells in which chromosomes are housed and replicated. In most...

SYNONYM:
Glomerular podocyte,

DEFINITION:
Podocytes (or visceral epithelial cells) are cells in the glomerular capsule in the kidneys that wrap around the capillaries of the glomerulus. The Bowman's capsule filters blood, holding back large molecules such as proteins, and passing through small molecules such as water, salts, and sugar, as the first step in forming urine.

URI:
http://cellfinder.de/ontology?uri=FMA_70967

Figure 11: Semantic Body Browser: podocyte

At the point of this writing the Semantic Body Browser incorporates 15 views for the human and mouse each. These views branch off into ten different levels of resolution and five developmental stages (Figure 12).

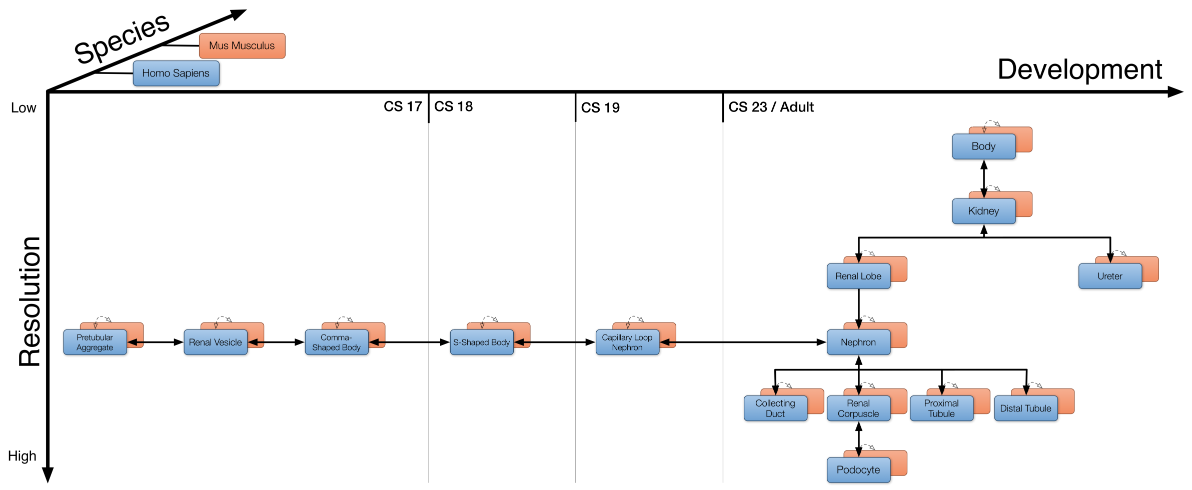


Figure 12: Browse tree

The blue and orange boxes illustrate the available views. It is possible to switch between the species at any time. The name for corresponding views in human and mouse are always the same, thus their names are only displayed on the blue labels.

4.2 Implementation

The following passages describe the technical implementation of the Semantic Body Browser in detail, giving code examples and explanations.

4.2.1 SVG to JSON Converter

Raphaël is used for the illustration and manipulation of SVG. It is capable of drawing circles, ellipses, rectangles and paths. This is realizes by calling appropriate functions with a couple of parameters that define the shape. As of Raphaël's nature the parameters are simple JavaScript objects.

As a consequence, SVG files need to be converted into JSON. The SVG to JSON converter is a command-line tool written in Python. It converts either a single or multiple SVG files into JSON. The tool needs two parameters: the input path and the output path (Listing 4).

```
1.  svn_to_json.py <PATH_TO_SVG> <OUTPUT>
```

Listing 4: SVG to JSON converter: prompt

The SVG to JSON converter requires Python to be installed. Moreover, the converter needs to be executable. If so, the tool can be used as demonstrated in line 1. The two parameters – written in uppercase and encapsulated in angled brackets – are mandatory. The first parameter can either point to a single file or a directory. In the latter case all SVG files in the given directory will be converted.

The conversion is achieved by looping over each tag of the SVG file and checking whether it is a supported shape or not. If it is supported all relevant attributes of the shape are stored. Apart from the actual shapes the group memberships are considered as well and stored in an extra array (Listing 5). Moreover the original dimensions of the view box are kept too. It is essential for the application to know which zoom level uses the space most efficient.

```
for i in range(1, len(svg)):
  tag = re.search('<(\/?)([a-z]+)', svg[i]).groups()
  id = re.search('id="([a-zA-Z0-9_]+?)(_[0-9]_)??"', svg[i])
  id = id.groups()[0].replace('_', '-') if id else False
  attrs = re.findall('([a-z-]+)="(\S+)"', svg[i])
  if len(group) and len(tag[0]):
    if tag[1] == raphaelGroup:
      group.pop()
  else:
    if tag[1] in raphaelElems:
       data['elements'].append({ 'type': tag[1] })
      for attr in attrs:
        if attr[0] in raphaelAttrs:
          data['elements'][-1][attr[0]] = attr[1]
        if id:
          data['elements'][-1]['id'] = id
      if group:
        data['groups'][group[-1]].append(len(data['elements']) - 1)
    elif tag[1] == raphaelGroup:
      if id:
        if len(group):
          data['groups'][group[-1]].append(id)
```

```
23.         group.append(id)
24.         data['groups'][id] = []
```

Listing 5: SVG to JSON converter: algorithm extract

The listing shows the main algorithm of the SVG to JSON converter. The script loops over each line of the SVG file and checks whether it contains a valid element or a group. If so the element, respective the group, is stored in an object. Lines 2 – 5 extracts the elements attributes. Next, the algorithm looks for closing group elements (lines 6 – 9). If a supported element is found (line 10), it is stored (line 11) together with its attributes (lines 12 – 16) and added to the active group if possible (lines 17 – 18). Lines 19 – 24 watch for opening group elements.

The JSON output contains three objects: elements, groups and viewBox. The object unit is a list of all shapes that makes up the illustration. Groups itself is an associative array which assigns each group a list of shapes or groups. Finally the viewBox simple contains information about the dimensions of the illustration (Listing 6).

```
{
    "elements": [
        {
            "height": "200",
            "width": "500",
            "stroke": "#000",
            "y": "100",
            "x": "100",
            "stroke-width": "20",
            "type": "rect",
            "fill": "#fff"
        },
        …
    ],
    "viewBox": {
        "width": 700,
        "height": 400
    },
    "groups": {}
}
```

Listing 6: SVG to JSON converter: output
The listing shows an extract of the output of the example SVG from listing 1. The rectangle is now stored in JSON notation and can be used by Raphaël.

4.2.2 Database

The application incorporates 30 views containing 395 units in total. Equally browsing the human and mouse is possible along ten different levels of resolution and five developmental stages (Figure 16). To ensure that all units are mapped to CELDA new terms have to be created that were either missing in human or mouse (Table 3). The 23 Carnegie stages are used to classify different developmental stages. As their resolution is too low for describing the right order of the first three developmental stages of the nephron, the 17th Carnegie stage has been subdivided. It should be noted that the Semantic Body Browser uses the Carnegie stages only, as the system provides a general classification of the developmental chronology of vertebrate embryos. Species-specific developmental stages can be translated with the conversion table of 4DXpress[16] [Haudry, 2012.]

Apart from that, the database does already contain 9 views and 102 units for the liver, gallbladder and generic blood vessels, which are likely to be integrated in a next step.

Label	Species	URI
pretubular aggregate	general	CELDA_000001468
	human	CELDA_000001474
interlobar artery	general	CELDA_000001462
	human	CELDA_000001463
	mouse	CELDA_000001464
interlobar renal vene	general	CELDA_000001422
	human	CELDA_000001428
	mouse	CELDA_000001429
mucosal fold	general	CELDA_000001438
	human	CELDA_000001439
	mouse	CELDA_000001440

[16] http://4dx.embl.de/4DXpress/reg/all/prepareSearch/mapping/stage.do

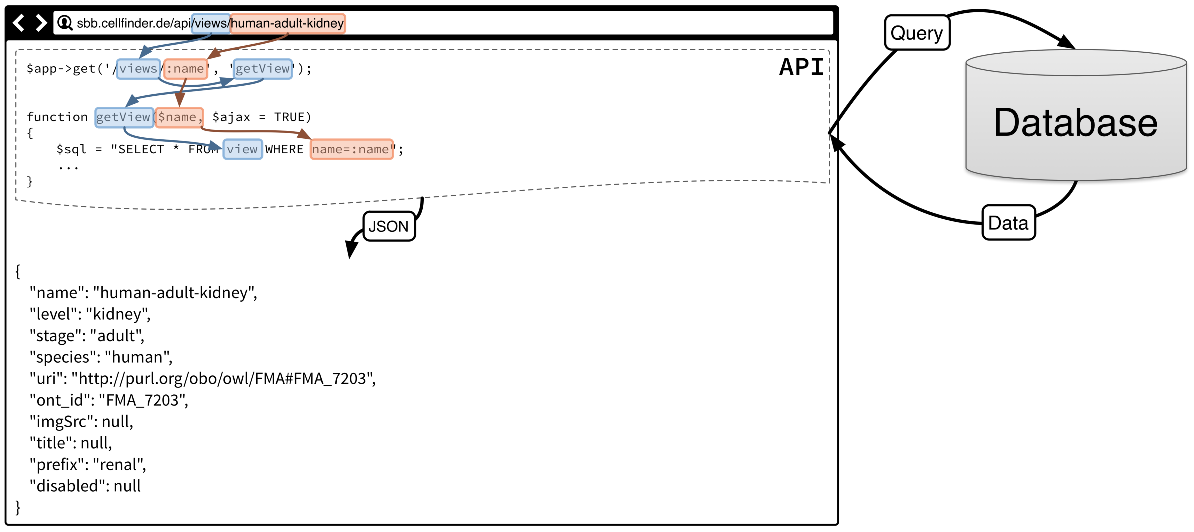
perivascular fibrous capsule	general	CELDA_000001442
	mouse	CELDA_000001442
nephrogenic interstitium	general	CELDA_000001427
	human	CELDA_000001431
primary process	general	CELDA_000001443
proximal cleft	general	CELDA_000001423
	human	CELDA_000001424
	mouse	CELDA_000001426
distal cleft	general	CELDA_000001444
	human	CELDA_000001445
	mouse	CELDA_000001446
ureteric trunk	general	CELDA_000001435
	human	CELDA_000001436
	mouse	CELDA_000001437
ureteral lumen	general	CELDA_000001447
	mouse	CELDA_000001448
lumen of distal tubule	general	CELDA_000001449
	human	CELDA_000001450
	mouse	CELDA_000001451
lumen of proximal tubule	general	CELDA_000001454
	human	CELDA_000001455
	mouse	CELDA_000001456
renal lobe	general	CELDA_000001452
	mouse	CELDA_000001453
visceral layer of renal corpuscle	general	CELDA_000001457
	human	CELDA_000001458
	mouse	CELDA_000001459
longitudinal muscle	general	CELDA_000001465
	human	CELDA_000001466
	mouse	CELDA_000001467

Table 3: Novel ontology terms

Over the course of developing the Semantic Body Browser, 41 new ontology terms have been created to ensure that all units are properly annotated.

4.2.3 API

The API is build with the PHP framework Slim. The application itself consists of one file only, the index.php. The deployment of the data contained in the database requires twelve methods (Table 2). Because the Semantic Body Browser does not need to change information during the runtime, all methods solely retrieve information from the database. In Slim a method needs to register as a “get” methods to listens for a particular input. The input is provided in form of a URL, which follows a certain pattern (Figure 13).



```
$app->get('/views/:name', 'getView');

function getView($name, $ajax = TRUE)
{
    $sql = "SELECT * FROM view WHERE name=:name";
    ...
}
```

```
{
   "name": "human-adult-kidney",
   "level": "kidney",
   "stage": "adult",
   "species": "human",
   "uri": "http://purl.org/obo/owl/FMA#FMA_7203",
   "ont_id": "FMA_7203",
   "imgSrc": null,
   "title": null,
   "prefix": "renal",
   "disabled": null
}
```

Figure 13: RESTful API

The Figure illustrates the information flow and implementation of the getView method (Table 1). The Structured Query Language (SQL) query is composed of the two last parameters of the URL. The first keyword “views” – highlighted in blue – defines which table is queried and the second keyword “human-adult-kidney” binds to the name. The API processes and sends this query to the database. The response is then printed in JSON.

Querying of the database is realized with prepared statements. A prepared statement’s characteristic is that the statement is evaluated by the database first and then binds certain values to it (Listing 7). This avoids security risks by SQL injections [PHP, 2012]. The structure of all methods is similar. First of all, the SQL statement is defined. Encapsulated in a try and catch block, the API then establishes a connection to the database server. After that the SQL statement is transmitted to get prepared. And just after that the given parameters are bond and the query is executed. The results are either returned to the parent method or printed on the screen. In both ways the results are first encoded into JSON to make them accessible for the JavaScript application.

```
function getView($name, $ajax = TRUE) {
  $sql = "SELECT * FROM view WHERE name=:name";
  try {
    $db = getConnection();
    $stmt = $db->prepare($sql);
    $stmt->bindParam("name", $name);
    $stmt->execute();
    $view = $stmt->fetchObject();
    $db = null;
    if ($ajax) {
      echo json_encode($view);
    } else {
      return json_encode($view);
    }
  }
  catch(PDOException $e) {
    echo '{"error":{"text":'. $e->getMessage() .'}}';
  }
}
```

Listing 7: API: getView implementation

4.2.4 The JavaScript Application

The application is build with AngularJS and using the AngularJS Seed as a template. Over the course of the development the overall file structure remained, only a folder for fonts has been added. In the following passages the different parts of the application are explained in detail.

4.2.4.1 Bootstrapping

The application is bootstrapped right after all script files are loaded asynchronously. General dependencies as well as the router are defined in js/app.js. AngularJS is being told that the application depends on services, filters and directives. This causes AngularJS to load the needed modules.

Furthermore a router is registered to listen for a certain location. It distinguishes between three types of locations to load the appropriate controllers and views (Table 4).

URL Pattern	Template	Controller	Description
/	home.html	AboutCtrl	Index or default page.
/about	about.html	AboutCtrl	Gives information about the application and the project.
/<VIEW>	view.html	ViewCtrl	Is used for displaying and manipulating the illustrations.

Table 4: JS application: router
Following the URL patterns, which are relative to the applications location, will invoke the related controllers and render the given templates. <VIEW> declares a parameter that represents a views name, e.g. "human-adult-body".

4.2.4.2 Controllers

The core of the application are the controllers, they contain the business logic and thus process the inputs and induce the display of the output. The Semantic Body Browser implements two controllers, one for displaying the illustrations and handling the browsing and another one for the about section. The latter contains only one function for changing the location and hiding an email address, due to the fact that the about section solely displays static texts and images.

The main controller, called ViewCtrl, controls the data flow while browsing. When the user enters a path that is different from "about", the ViewCtrl will be loaded. Before it is executed AngularJS ensures that all dependencies are loaded. These dependencies are needed by the controller to perform all its tasks. There are two different groups of dependencies; dependencies prefixed with a dollar sign *$* belonging to AngularJS and everything else which are custom extensions.
Once the dependencies are injected, the controller will load information about the current view from the database. It does so by calling a custom service and passing specific parameters. When the data is successfully loaded AngularJS will execute a callback function, which processes the data in a way that it can be used in the view (Listing 8).

```
1. Api.query({
2.   a: 'search',
3.   b: $scope.data.view.stage,
4.   c: $scope.data.view.species
5. }, function(data) {
6.   $scope.zoomViews = data;
7. });
```

Listing 8: JS application: invoking a service

The example demonstrates how a service is called in AngularJS. The service “Api” queries the API (line 1) and passes three parameters a, b and c to it (lines 2 – 4). These parameters are used for defining the database query. When the API returns data the callback function (line 5) will set the zoomViews variable to the returned data.

Information about the current view, the integrated units and the availability of higher resolutions are fetched. Furthermore, the illustration is loaded. The data is stored as an attribute of the current scope to make it accessible by the view.

Apart from that, the controller provides a number of methods to get or set variables. These are needed to make certain information available for views because those views have reduced possibilities to process data (Table 5).

Method	Parameters	Description
prefix	hasPrefix	Returns a views prefix depending on the parameter.
isZoomable	unit	Returns whether a higher resolution for the given unit is available.
getOntId	unit	Returns the ontology identifier of a unit.
getDefinitions	unit single	Returns a single or multiple (depending on the parameter single) definitions of a unit.
setActiveUnit	unit group	Call the services News to broadcast and set a new active unit.
setView	view	Changes the illustration to the given view if available.
setLocation	url	Changes the URL according to the parameter.

Table 5: JS application: controller methods

Finally, the controller watches for changes of the current location and the active unit. The benefit is that changes in any of the two are immediately applied to the other one. This so called two-way data binding is currently not implemented between the location service and a variable by AngularJS and thus has to be set up manually. Because the active unit can be changed outside of the controller, it is being watched for changes in the local variables and listened for changes from elsewhere (Listing 9).

```
1. $scope.$on('activeUnit', function() {
2.   $scope.activeUnit = News.activeUnit;
3. });
4. $scope.$watch(function(){
5.   return $location.search().unit;
6. }, function(newValue, oldValue) {
7.   $scope.setActiveUnit(newValue);
8. });
```

Listing 9: JS application: watcher and listener

A listener is set by the method $on, defining what should be listened for as well as the consequences (line 2). In this case the controller listens for the activeUnit and sets the scope's variable activeUnit to the one of the service News. The event activeUnit is broadcasted by the service News whenever the active unit changes. This way any controller or directive can act upon the change. A watcher does only detect changes of a local variable; the unit parameter of the URL in this case e.g. "./adult-human-body?unit=<PARAMETER>" (lines 4 – 5). The call back function (lines 6 – 8) passes two values, the new and the old one of the variable that is being watched. Here it just calls the method setActiveUnit and passes the new active unit.

4.2.4.3 Services

As described in the previous chapter, the view controller ViewCtrl loads several dependencies. All of the custom dependencies are services. A service is defined as a singleton that carries out a specific task [AngularJS, 2012].

The Semantic Body Browser implements three services, one for querying the API, a second one for loading the illustrations and a third one that delivers information across different parts of the application.

The first and the second service technically do the same and only distinguish in their data source, which they are querying. Both depend on a module that is called

$resource and is provided by AngularJS. The $resource service dramatically simplifies the interaction with a RESTful API, like the one the Semantic Body Browser implements (Listing 10).

```
1.  angular
2.    .module('sbb.services', ['ngResource'])
3.    .factory('Api', ['$resource', function($resource){
4.      return $resource('../api/:a/:b/:c/:d', {}, {});
5.    }]);
```

Listing 10: JS application: service

Lines 1 and 2 register a new model called sbb.services and declare its dependencies: ngResource, another module. Modules are simply the building blocks of an application, which can depend on each other. The factory method in line 3 defines a new service called Api. This service only needs to invoke and return the method $resource and define which parameters – prefixed with a colon – are available (line 4).

The third service called News broadcasts events across the application. Because the Semantic Body Browser application is composed of several directives and scopes, it can be tricky to keep them all synchronized. The News service takes care of that. It can be injected in any part of the application and provides a global way of storing objects, plus each time an object is updated the news will be broadcasted to anyone who is listening. The service News can broadcast three types of information: first changes to the active unit, second the happening of a global click and its target element and third the height of the information panel in the side bar.

4.2.4.4 Directives

The Semantic Body Browser defines eleven directives. These directives are used to enhance normal HTML to be able to perform more interactive tasks. Five out of the eleven directives are responsible for providing certain navigation options in the header section. Another four are needed to make the side bar responsive, one handles manipulation of SVG and the last directive works on the global scope to detect certain events (Table 6).

Directives	Description
globalEvents	Registered a global eventListener for a mouse click.
about	Handles the about drop-down menu.
bookmarks	Handles the bookmark drop-down menu as well as storing, retrieving and opening a bookmark.
species	Handles the species drop-down menu as well as displaying the available species for a certain view.
stage	Handles the developmental stage drop-down menu and processes the different developmental stages.
zoom	Handles the zoom drop-down menu and processes the different zoom views.
raphael	Display illustrations and processes all inputs and interactions with the illustration.
units	Changes the height of the container units according to the information panels below.
accordion	Enables the information panels to expand and collapse.
omeSlider	Enables the picture slider.
omeLoader	Loads the picture from the Omero image server.
browserDim	Only for development: Shows the browsers dimensions.

Table 6: JS application: directives

All directives share a similar structure. They define a name and list certain dependencies if existing. The main logic is returned as an object that holds all of the necessary information. It has to be defined how the directive will be invoked by the HTML, what the template and scope looks like and the main functionality (Listing 11).

```
.directive('zoom', ['$location', 'News', function($location, News) {
  return {
    restrict: 'C',
    templateUrl: 'partials/zoom.html',
    scope: {
      setLocation: '&',
      zoomViews: '=',
      level: '@'
    },
    link: function(scope, element) {
```

```
11.       var opened = true;
12.       scope.enabled;
13.       scope.toggle = function(state, init) {
14.         if (scope.enabled || init) {
15.           opened = (typeof state === 'undefined') ? !opened : state;
16.           element.removeClass(opened ? 'closed' : 'opened');
17.           element.addClass(opened ? 'opened' : 'closed');
18.         }
19.       }
20.       scope.$on('click', function() {
21.         if (element.find(News.clickTarget.tagName)[0] !== News.clickTarget) {
22.           scope.toggle(false);
23.         }
24.       });
25.     }
26.   }
27. }])
```

Listing 11: JS application: directive

The listing shows the shortened source code of the directive Zoom but the main characteristics stay the same. The directive is defined in line 1 and has two dependencies: the $location and News service. The returned object (line 2) contains several settings and a linking function. This directive is called by placing its name inside a class attribute (line 3). The template that is used – a drop-down menu – is loaded from an external file in the partials folder (line 4). Every directive has its own scope. This makes it independent from a controller, avoids conflicts and allows the directive to be used in different environments. This zoom directive is though linked with a controller, as it needs a couple of data. First of all, the method setLocation (line 6) is connected to this scope. Calling it from within the scope will fire the external method. This is useful, as only one method needs to be maintained. Apart from that is the object zoomView is connected to this scope as well (line 7). There is no direct need for it but simple binding, as it is realized with the level object, only works with primitive data types and not with objects. The level is only a representation of the controller's variable and changes to it do not affect the parent scope (line 8). The link function invokes the behaviour of the directive. It has two important parameters: scope and element (line 10). The "scope" is a reference of the directives scope and the "element" refers to the DOM element, on which the directive is applied. In lines 11 – 12, two variables are set with the important difference that the first one is a private variable and cannot be accessed from elsewhere, while the second is public because it is an attribute of the scope.

The toggle function for opening and closing the drop-down menu is defined over the lines 13 – 19. Lines 20 – 24 show the use of the service News. This directive listens for a global click. When such a click is broadcasted it will check its target and close the menu in case the click was performed outside of the drop-down menu.

The most complex directive handles the illustration and provides ways for its manipulation. First the data provided by the database is used to render the illustration. The directive furthermore takes care of the right scaling by adjustments to the view box and the stroke size. Besides that, a number of different mouse and touch events are registered to make interactions available (Table 7).

Event	Description
mouseover mouseout	Pointing the mouse cursor over a unit highlights it.
mousedown mousemove mouseup	The illustration can be dragged while the mouse is pressed.
dragstart drag dragend	Equivalent touch events to mousedown, mousemove and mouseup.
click	A mouse click on a unit highlights the unit and the related button in the unit list in the right side bar.
tap	Equivalent touch event to the mouse click.
double click	A double click on a unit launches a dialog which gives a definition of the unit and the possibility to further zoom in (if available) or to reveal more information on CellFinder.
double tap	Equivalent touch event to the double click.
mouse wheel	The illustration can be magnified or shrunken by using the mouse wheel.
pinch	Equivalent touch event to the mouse wheel but magnification is triggered by a so-called pinch gesture. Two fingers have to be placed on the illustration. Reducing the distance of the two fingers shrinks the illustration and increasing the distance magnifies it.

Table 7: JS application: user events

The table lists all events that the directive raphael listens for.

4.2.4.5 Templates

The view that is displayed in the browser is composed of templates. Templates consist of HTML, CSS and are combined with AngularJS specific elements and attributes.

The Semantic Body Browser separates several parts of a template in sub templates, located in the partials directory, to make them reusable. The three main templates relate to the three locations: index/home, about, views.

All main templates follow a certain structure. They comprise of a header and a main container. The header contains the general navigation whereas all of the content is part of the main container. The difference between the three templates lies in the nature of their content. Two of them, the home screen and the about section, only display static information. The template for the graphical browsing instead has to handle dynamic data and thus is much more complex (Listing 12).

```
<header>
  <nav id="nav">
    <ul>
      <li class="about drop" set-location="setLocation(url)"></li>
      …
    </ul>
  </nav>
  <form id="search">
    <div class="wrapper">
      <input ng-model="searchInput" name="search" type="search"
placeholder="Filter Units" />
    </div>
  </form>
  <h1 class="right">{{title}}</h1>
</header>
<div id="sidebar">
  <section id="units" units>…</section>
  <div id="information" accordion>…</div>
</div>
<div id="main" class="right">
  <raphael id="raphael"
           ilu="ilu"
           set-view="setView(view)"
           set-active-unit="setActiveUnit(unit, group)"
```

```
                is-zoomable="isZoomable(unit)"
                get-ont-id="getOntId(unit)"
                get-definitions="getDefinitions(unit, onlyOne)" />
21. </div>
```

Listing 12: JS application: template

The listing shows an extract of the template view. It features the top bar (lines 1 – 14), side bar (lines 15 – 18) and the illustration panel (lines 19 – 21). The template is composed of HTML5 and AngularJS specific attributes and elements – highlighted in orange.

4.2.5 Grunt

To reduce the overall size of the application Grunt concatenates and minifies most files. In detail, all JavaScript files that are part or belong to the same framework as well as Cascading Style Sheets (CSS) are concatenated and then minified. The main advantage is that the application can load much faster which is essential for mobile usage (Table 8).

Source Files	Concatenated File	Uncompressed (Byte)	Minified & Compressed (Byte)
jquery.custom.js jquery.mousewheel.js jquery.omeloader.js jquery.omeslider.js hammer.js jquery.hammer.js	jquery.min.js	234.990	**29.360**
raphael.js	raphael.min.js	218.398	**30.982**
app.js controller.js directives.js services.js filters.js	app.min.js	31.751	**4.709**
normalize.css style.css drop.css	style.min.css	35.823	**5.113**

ie9.css	ie9.min.css	577	**221**
ie.css	ie.min.css	587	**273**
Total		522.126	**70.658**

Listing 8: JS application: code compression

Compression of code can have a significant impact on the overall size of an application. Grunt minifies the code by removing unnecessary white space and refactoring. Additionally, server-side code compression that is performed by GZip does reduce the overall file size further. In total the code has been reduced by ~86% from ~522 KB to ~70 KB. (Compression has been performed on the 9 Oct. 2012)

4.3 Technical Requirements

The Semantic Body Browser is a web-based application. It is therefore necessary to have access to the Internet and to use a modern web browser [Lazaris, 2012]. The application can be used with many computer platforms, smart phones and tablets. The application has been tested extensively (Table 9).

Browser	Version	Level	Limitations
Chrome	11+	A	
	5+	B	• Dragging the illustration leads to offsets. • Deselection of highlighted units is not possible • Microscope pictures can't be displayed
Safari	5+	A	
	4+	B	• Dragging the illustration leads to offsets. • Deselection of highlighted units is not possible • Microscope pictures can't be displayed
Mobile Safari	iOS 5.1+	A-	• Magnifying the illustration is slow
Firefox	3.6+	A	
Opera	11.6+	A	
Internet Explorer	9+	A-	• Interaction is slower compared to other browsers.

Table 9: Compatibility

The experience levels vary from A to C. If the browser is capable of delivering all technical and visual features it is labelled with A. B comprises browsers that only have minor visual limitations. Browsers of level C show significant limitations but can still be used. The browsers are sorted by the recommended usage in respect to the Semantic Body Browser.

5 Discussion

The Semantic Body Browser is a web-based tool to graphically explore an organism's body and obtain comprehensive biological information on the way. It is part of the CellFinder project and therefore mainly focuses on students and researchers who operate in cell and molecular biology or medicine.

5.1 Benefits

It could be argued that there already exist similar applications[17] [18] [19] [20] but the Semantic Body Browsers strongly differentiates itself in a number of features. First of all the Semantic Body Browser is a web based solution. As a consequence, any device that has access to the Internet and fulfils the technical requirements can run the application. Only the Zygote Body is web based too. Furthermore the Semantic Body Browsers relies on two-dimensional abstracted illustrations. While none of them claim to describe the reality in every shape, they concentrate on main anatomical structures and biological functions of the displayed area. This simplifies the illustration and helps the understanding. As the main goal is not to provide an anatomical replica we think minor inaccuracies do not harm the process of navigation through an organism's body but rather make it easier by ensuring a minimal size for units and accessibility of every unit without rotation. Another distinction is that the Semantic Body Browser integrates more than one dimension. Most application solely provides the manipulation of the resolution whereas the Semantic Body Browser has the possibility to switch between different species and trace developmental stages. But the unique characteristic that distinguishes the Semantic Body Browser from all other tools is the underlying CELDA ontology. By means of annotated vector graphics, the use an ontology and the integration into CellFinder, it is possible to deliver relevant and detailed biological data along the navigation. The components or units of the Semantic Body Browser's illustrations

[17] http://www.zygotebody.com
[18] http://www.imaios.com/en/iPad-iPhone-Android
[19] http://www.pocketanatomy.com
[20] http://www.modality.com/body/

express a meaning, which enhances with the on going development on CellFinder and respectively CELDA.

To conclude, there are a number of human body browsers available but they do only focus on a single domain and this undoubtedly quite well. The Semantic Body Browser on the other hand is a web based tool that works across several devices and provides comprehensive biological data in a wide range of domains while navigation along the three dimensions: resolution, development and species.

5.2 Semantics

Another issue to discuss is why this application is called semantic. The semantics expresses the meaning of two related objects. Within the scope of the web, it often refers to the annotation of information in a way that the information can be processed and - in best case - understood by machines. In this respect, the Semantic Body Browser is not directly semantic. The semantics reveals when the application is seen as a part of CellFinder. All of CellFinder's data is linked to the CELDA ontology and so are the illustrations of the Semantic Body Browser. Every view and every unit is mapped to a certain term. This way the illustrations as well as its components express meaning.

CELDA already correlates most of the terms by their hierarchy (is_a), development (develops_form, develops_into) and species (via the general Uberon ontology). Therefore it could be asked why the application uses an own data source and does not fully rely on CELDA. The problem in relation to the Semantic Body Browser is the high resolution and the diverse nature of the different ontologies (Figure 14). It does not always make sense to provide a new view / illustration for every term and would furthermore exceed the budget. Thus a mapping is needed which defines relevant and useful steps for the browsing experience. For this reasons a database is sufficient. Nevertheless, as the development of CellFinder and CELDA moves forward it could be discussed if it is beneficial to develop a bridge-ontology to reflect more semantics.

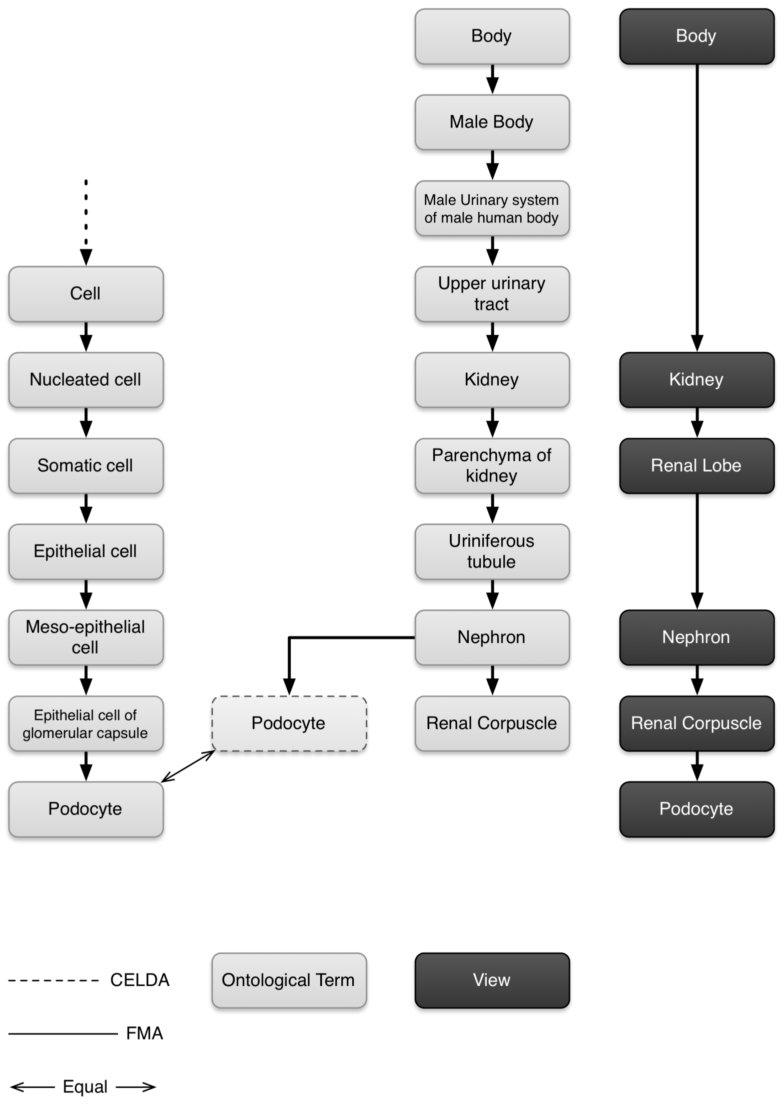


Figure 14: Comparison of ontology levels with the Semantic Body Browser's views
The grey ontological terms represent an abstract from the Foundational Model of Anatomy (FMA) ontology. Three difficulties occur in this example. First, the resolution is very high and it would not enhance the browsing experience to include views for every term. Second, the cellular level is not always represented ideally. The podocyte of the FMA occurs at a totally different location and CELDA maps it not accurately enough. The third difficulty is that the integration of several ontologies under CELDA leads to multiple terms of the same object. As they are already classified as equal, the question would be which term should be used for browsing? This cannot be answered in general and depends on the situation. To conclude, it is better to create an own, browsing-specific hierarchy and map the views to CELDA.

5.3 Differences Between Species

Within the scope of this thesis, the Semantic Body Browser implements 30 views which split up equally for each species: human and mouse. However, only a single illustration of the mouse differs from the human ones, which is the body's overview. This is caused by the fact that the kidneys anatomy does not show significant anatomical differences. The higher the resolution, the higher is the similarity between the human and mouse anatomy.

Including species, whose common origin is further away, would surely lead to more differences, which would result in more species-specific illustrations.

5.4 Monochrome Illustrations

The Semantic Body Browser only displays monochrome illustrations. While it is true that colours help to recognize and distinguish shapes it can get distracting. Furthermore, the contrast between objects vanishes the more colours are used. In respect to biological illustration it should also be noted that some of the commonly used colours are arbitrary. Especially sub cellular colouring derives from specific staining.

In conclusion, consistently colouring biological illustrations needs comprehensive research, which was out of the scope of this thesis. It was therefore decided to use monochrome graphics first and re-evaluate colouring in a later step.

6 Outlook

The scope of this thesis was to develop the Semantic Body Browser with all technical features and to demonstrate them as a proof of principle for one organ in two species. Future updates could focus on the extension of views. More organs, species or developmental stage would enhance the user's browsing experience. As mentioned in the previous chapter most anatomical differences vanish between mammals as the resolution is increased. It would therefore be interesting to integrate species whose biological origin is further away from each other.

Another interesting field of research could involve the integration of processes. Many of an organ's function originate in cellular or sub cellular processes. Showing or linking these processes within the illustration could help to make them easier to understand. Creating animations surely takes time but fortunately, the tool used for displaying the vector graphics in the Semantic Body Browser already satisfy the technical requirements.

In general, enhancing the annotation of the illustrations and integrating more interactions with CellFinder's data would be appealing and surely help CellFinder to improve the user experience.

7 Reference

[AngularJS, 2012]	AngularJS, *Developer Guide*, AngularJS, 2012. [Online]. Available: http://docs.angularjs.org/guide/ [Accessed: 8 Aug. 2012].
[CellFinder, 2012]	CellFinder team, *About*, CellFinder, 2 Oct. 2012. [Online]. Available: http://cellfinder.de/beta/about [Accessed: 2 Oct. 2012].
[Crockford, 2001]	D. Crockford, JavaScript: The World's Most Misunderstood Programming Language, Douglas Crockford, 2001. [Online]. Available: http://javascript.crockford.com/javascript.html [Accessed: 2 Oct. 2012].
[Davidson, 2009]	A.J. Davidson, Mouse kidney development, *StemBook*, ed. The Stem Cell Research Community, 15 Jan. 2009. [Online]. Available: doi/10.3824/stembook.1.34.1 [Accessed: 5 Sep. 2012].
[Gerett, 2005]	J.J. Gerett, Ajax: A New Approach to Web Applications, Adaptive Path, 18 Feb. 2005. [Online]. Available: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications [Accessed: 29 Sep 2012].
[Guyton, 2006]	A. Guyton and J. Hall, *Textbook of Medical Physiology*, 11th ed. Philadelphia: Elsevier Saunders, 2006, p. 310.
[Harding, 2011]	S.D. Harding et al., The GUDMAP database – an online resource for genitourinary research, *Development*, vol. 138, Jul. 2011, pp. 2845-2853. [Online]. Available: doi: 10.1242/dev.063594 [Accessed: 14 Aug. 2012].
[Haudry, 2008]	Y. Haudry et al., 4DXpress: a database for cross-species expression pattern comparisons, *Nucleic Acids Res.*, vol. 36 (Database issue), Jan 2008, pp. D847-D853. [Online]. Available: doi: 10.1093/nar/gkm797 [Accessed: 5 Oct. 2012].
[Lazaris, 2012]	L. Lazaris, *Old Browsers Are Holding Back The Web*, Smashing Magazine, 9 Jul. 2012. [Online]. Available: http://www.smashingmagazine.com/2012/07/09/old-browsers-are-holding-back-the-web/ [Accessed: 27 Sep. 2012].
[Manski, 2012]	D. Manski, *Anatomy of the kidney: gross anatomy*, Urology Textbook, 11 Oct 2012. [Online]. Available: http://www.urology-textbook.com/kidney-anatomy.html [Accessed: 12 Oct. 2012].
[McMahon, 2008]	A.P. McMahon et al., GUDMAP: the genitourinary developmental molecular anatomy project, *J Am Soc Nephrol.*, vol. 19, no. 4, 20 Feb. 2008, pp. 667-671. [Online]. Available: 10.1681/ASN.2007101078 [Accessed: 14 Aug. 2012].

[Mozilla, 2012]	Mozilla Developer Network and contributors, *JavaScript*, Mozilla Developer Network, 24 Sep. 2012. [Online]. Available: https://developer.mozilla.org/en-US/docs/JavaScript [Accessed: 30 Sep. 2012].
[Pavenstädt, 2003]	H. Pavenstädt, W. Kriz, M. Kretzler, Cell Biology of the Glomerular Podocyte, *Physiological Reviews*, vol. 83, no. 1, 2003, pp. 253-307 [Online]. Available: doi: 10.1152/physrev.00020.2002 [Accessed: 11 Aug. 2012].
[PHP, 2012]	PHP contributors, *Prepared statements and stored procedure*, PHP, 5 Oct. 2012. [Online]. Available: http://php.net/manual/en/pdo.prepared-statements.php [Accessed: 9 Oct. 2012].
[Reenskaug, 1979]	T. Reenskaug, "MODELS - VIEWS - CONTROLLERS", Xerox Parc: Palo Alto, 10 Dec. 1979. [Online]. Available: http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf [Accessed: 2 Oct. 2012].
[Resig, 2009]	J. Resig, *Talk: The DOM is a Mess*, John Resig, 2 Feb. 2009. [Online]. Available: http://ejohn.org/blog/the-dom-is-a-mess/ [Accessed: 17 Sep. 2012].
[Schiller, 2011]	J. Schiller, SVG Support, Codedread, 24 Mar 2011. [Online]. Available: http://www.codedread.com/svg-support.php [Accessed: 8 Oct 2012].
[W3C, 2011]	W3C, Scalable Vector Graphics (SVG) 1.1 (Second Edition), W3C, 16 Aug. 2011. [Online]. Available: http://www.w3.org/TR/SVG11/ [Accessed: 9 Aug. 2012].
[W3C, 2012]	W3C, Document Object Model (DOM) Technical Reports, W3C, 2 Jun. 2012. [Online]. Available: http://www.w3.org/DOM/DOMTR [Accessed: 18 Aug. 2012].
[W3Techs, 2012]	W3Techs, Usage of JavaScript libraries for websites, W3Techs – Web Technology Surveys 9 Oct. 2012 [Online]. Available: http://w3techs.com/technologies/overview/javascript_library/all [Accessed: 9 Oct. 2012].
[Werner, 2012]	S. Werner et al., poster presented at the BioIT World, Bosten, USA, 2012. [Online]. Available: http://cellfinder.de/beta/dl/Poster-CellFinder-BioIt-2012.pdf [Accessed: 12. Aug 2012].
[Wikipedia, 2012]	Wikipedia Community, *Kidney*, Wikipedia – The Free Encyclopedia, 13 Aug. 2012. [Online]. Available: http://en.wikipedia.org/wiki/Kidney [Accessed: 14 Aug. 2012].

APPENDIX

List of Abbreviations

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CELDA	Cell: Expression, Localization, Development, Anatomy
CS	Carnegie stage
CSS	Cascading Style Sheets
DB	Database
DOM	Document Object Model
FMA	Foundational Model of Anatomy
HTML	Hypertext Markup Language
JS	JavaScript
JSON	JavaScript Object Notation
KB	Kilobyte
PHP	PHP: Hypertext Preprocessor
REST	Representational State Transfer
SQL	Structured Query Language
SVG	Scalable Vector Graphics
TS	Theiler stage
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VML	Vector Markup Language
W3C	World Wide Web Consortium
XML	Extensible Markup Language

List of Figures

List of Listings

List of Tables

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben. Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Datum	Unterschrift