BlattWerkzeug Documentation

Release 0.1

Marcus Riemer

Nov 13, 2018

For Users

This guide is mainly technical documentation for developers, system administrators or advanced users that want to develop their own language flavors inside BlattWerkzeug. It is *not* a guide for pupils or other end users that want to know how to program *using* BlattWerkzeug.

Contrary to "normal" compilers, BlattWerkzeug only operates on abstract syntax trees. Every code resource (`SQL`, `HTML`, a regular expression, ...) that exists within BlattWerkzeug is, at its core, simply a syntaxtree. All operations that are described in this manual work with the syntaxtrees of these code resources in one way or another.

CHAPTER 1

Core Concepts

Conventional development environments are programs that are tailored to suit the needs of professionals. Due to their complexity they do not lend themselves well to introduce pupils to programming. BlattWerkzeug is a tool that is geared towards "serious learners" and is intended to be used with support from teachers or some similar form of supervision.

To eliminate the possibility of syntactical errors while programming, the elements of the programming- or markup-languages are represented by graphical blocks, similar to the approach taken by the software Scratch. These blocks can be combined by using drag & drop operations.

The current aim is to provide an environment for the following programming languages:

- `SQL` and databases in general. This explicitly includes the generation and modification of schemas.
- `HTML` and `CSS` to generate web pages.
- A `HTML`-dialect that supports basic algorithmic structures like conditionals and loops.
- A "typical" imperative programming language.
- Regular Expressions.

1.1 Block & Programming Languages

Fig. 1: Relations of syntaxtrees, block- and programming-languages.

At the very core, there are four different structures involved when a program is edited with a block editor:

- The **grammar** defines the basic structure of an abstract syntax tree that may be edited. It may be used to automatically generate block languages and validators.
- The **abstract syntax tree** represents the structure of the code that is via the **block editor**. In a conventional system this can be thought of as a "file".

- The block editor know how to represent the "file" because it uses a **block language** which controls how the syntaxtree is layouted and which blocks are available in the sidebar.
- The actual compilation and validation is done by a **programming language**.

For everyday users this distinction is not relevant at all. They only ever interact with "files" that make use of certain block languages.

Advanced users like teachers may adapt existing block languages (or even create entirely new ones) to better suit the exact requirements of their classroom. Especially removing functionality from block languages should be a relatively trivial operation. So having a variant of the SQL block language

The creation or adaption of existing block languages languages should be "easy", at least for the targeted audiences: Programmers with a background in compiler construction should "easily" be able to add new languages and teachers with a little bit of programming experience should "easily" be able to tweak existing languages to their liking.

1.2 Projects

All work in BlattWerkzeug is done in the scope of so called "projects". Projects are the main category of work and have at least a name and a user friendly description. Apart from that they bundle together various resources and assets such as databases, images and code.

CHAPTER 2

The Abstract Syntax Tree

In order to allow the creation of easy to use block editors, BlattWerkzeug needs to define its own compilation primitives (syntaxtrees, grammars & validators). The main reason for this re-invention the wheel is the focus of existing software: Usually compilers are focused on speed and correctness, not necessarily a friendly representation for drag & drop mutations. BlattWerkzeug instead focuses exclusively on working with a syntaxtree that lends itself well to be (more or less) directly presented to the end user. Typical compiler tasks that have to do with lexical analysis or parsing are not relevant for BlattWerkzeug.

The syntax tree itself is purely a data structure and has no concept of being "valid" or "invalid" on its own (this is the task of validators). It also has no idea how to it should "look like" in its compiled form (this is the task of block languages).

A single node in the syntaxtree has at least a **type** that consists of two strings: A local `name` and a `language`. This type is the premier way for different tools to decide how the node in question should be treated.

The `language` is essentially a namespace that allows the use of identical `name`s in different contexts. This is useful when describing identical concepts in different languages:

- Programming languages have some concept of branching built in, usually with a keyword called `if`. Using the `language` as a prefix, two languages like e.g. `Ruby` and `JavaScript` may both define their concept of branches using `if` as the `name`.
- Markup languages usually have a concept of "headings" that may exist on multiple levels. No matter whether the markup language in question is `Markdown` or `HTML`, both may define their own concept of a `heading` in their own namespace.

Nodes may define so called **properties** which hold atomic values in the form of texts or integers, but never in the form of child nodes. Each of these properties needs to have a name that is unique in the scope of the current node.

The children of nodes have to be organized in so called **child categories**. Each of these categories has a name and may contain any number of children. This is a rather unusual implementation of syntaxtrees, but is beneficial to ease the implementation of the user interface.

The resulting structure has a strong resemblance to an `XML`-tree, but instead of grouping all children in a single, implicit scope, they are organised into their own-subtrees.

2.1 JSON-representation and datatype definition

In terms of Typescript-Code, the syntaxtree is defined like this:

```
1  export interface NodeDescription {
2    name: string
3    language: string
4    children?: {
5      [childrenCategory: string]: NodeDescription[];
6    }
7    properties?: {
8      [propertyName: string]: string;
9    }
10 }
```

Lines 2 - 3: The type of the node As mentioned earlier: Both of these strings are mandatory.

Lines 4 - 6: Optional child categories A dictionary that maps string-keys to lists of other nodes.

Lines 7 - 9: Optional properties A dictionary that maps string-keys to atomic values, which are always stored as a string.

Syntaxtrees may be stored as `JSON`-documents conforming to the following schema (which was generated out of the `interface`-definition above): *Storage Formats*.

2.2 Visual & textual examples

The following examples describe one approach of how expressions could be expressed using the described structure. This series of examples is meant to introduce a visual representation of these trees and was chosen to show interesting tree constellations. It depicts a valid way to express expressions, but this approach is by no means the only way to do so!

The simplest tree consists of a single, empty node. You can infer from its name that it is probably meant to represent the `null`-value of any programming language.

```
{
  "language": "lang",
  "name": "null"
}
```

Fig. 1: Expression `null`

Properties of nodes are listed inside the node itself. The following tree corresponds to an expression that simply consists of a single variable named `numRattle` that is mentioned.

```
{
  "language": "lang",
  "name": "exprVar",
  "properties": {
    "name": "numRattles"
  }
}
```

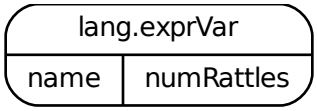


Fig. 2: Expression `numRattle`

Children of trees are simply denoted by arrows that are connecting them. They are grouped into named boxes that define the name of the child group in which they appear in. So the following tree represents a binary expression that has two child groups (`lhs` for "left hand side" and `rhs` for "right hand side") and defines the used operation with the property `op`.

```
{
  "language": "lang",
  "name": "expBin",
  "properties": {
    "op": "eq"
  },
  "children": {
    "lhs": [
      {
        "language": "lang",
        "name": "expVar",
        "properties": {
          "name": "numRattles"
        }
      }
    ],
    "rhs": [
      {
        "language": "lang",
        "name": "null"
      }
    ]
  }
}
```

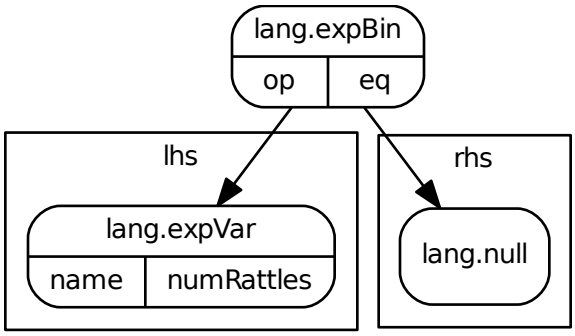


Fig. 3: Expression `null == numRattle`

CHAPTER 3

Programming Languages

As syntaxtrees may define arbitrary tree structures, some kind of validation is necessary to ensure that certain trees conform to certain programming languages. The validation concept is losely based on `XML Schema` and `RelaxNG`, the syntax of the latter is also used to describe the grammars in a user friendly textformat.

3.1 The traditional approach

A somewhat typical grammar to represent an `if` statement with an optional `else`-statement in a nondescript language could look very similar to this:

```
if         ::=   'if' <expr> 'then' <stmt> ['else' <stmt>]
expr       ::=   '(' <expr> <binOp> <expr> ')'
                 | <var_name>
                 | <val_const>
expr_list  ::=   <expr>
                 | <expr> ',' <expr_list>
                 | ''
stmt       ::=   <var_name> = <expr>
                 | <var_name> '(' <expr_list> ')'
```

This approach works fine for typical compilers: They need to derive a syntax tree from any stream of tokens. It is therefore important to keep an eye on all sorts of syntactical elements. This comes with its very own set of problems:

1. The role of whitespace has to be specified.
2. Some separating characters have to be introduced very carefully. This is usually done using distinctive syntactic elements that are not allowed in variable names (typically all sorts of brackets and punctuation).
3. Handing out proper error messages for syntactically incorrect documents is hard. A single missing character may change the semantics of the whole document. This implies that semantic analysis is usually only possible on a syntactically correct document.

3.2 The BlattWerkzeug approach

But BlattWerkzeug is in a different position: It is not meant to create a syntax tree from some stream of tokens but rather begins with an empty syntax tree. This frees it from many of the problems that have been mentioned above:

1. There is no whitespace, only the structure of the tree.
2. There is no need for separation characters, only the structure of the tree.
3. Syntax errors equate to missing nodes and can be communicated very clearly. Semantic analysis does not need to rely on heuristics on how the tree could change if the "blanks" were filled in.

This entirely shifts the way one has to reason about the validation rules inside of BlattWerkzeug: There is no need to worry about the syntactic aspects of a certain language, the "grammar" that is required doesn't need to know anything about keywords or separating characters. Its sole job is to describe the structure and the semantics of trees that are valid in that specific language.

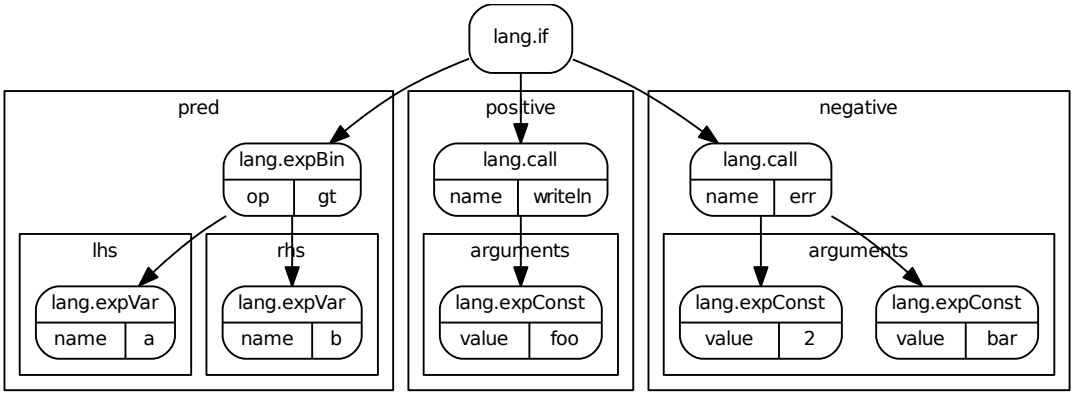
3.2.1 Example AST: `if`-Statement

The following example further motivates this reasoning. An if statement can be described in terms of its structure and the underlying semantics: It uses a `predicate` to distinguish whether execution should continue on a `positive` branch or a `negative` branch.

This is a possible syntaxtree for an `if` statement in some nondescript language that could look like this:

```
if (a > b) then
  writeln('foo')
else
  err(2, 'bar')
```

In BlattWerkzeug, an if statement could be represented by using three child groups that could be called `predicate`, `positive` and `negative`. Each of these child groups may then have their own list of children.



Now lets see what happens if the source is invalidated by omitting the `predicate` and the `then`:

```
if
  writeln('foo')
```

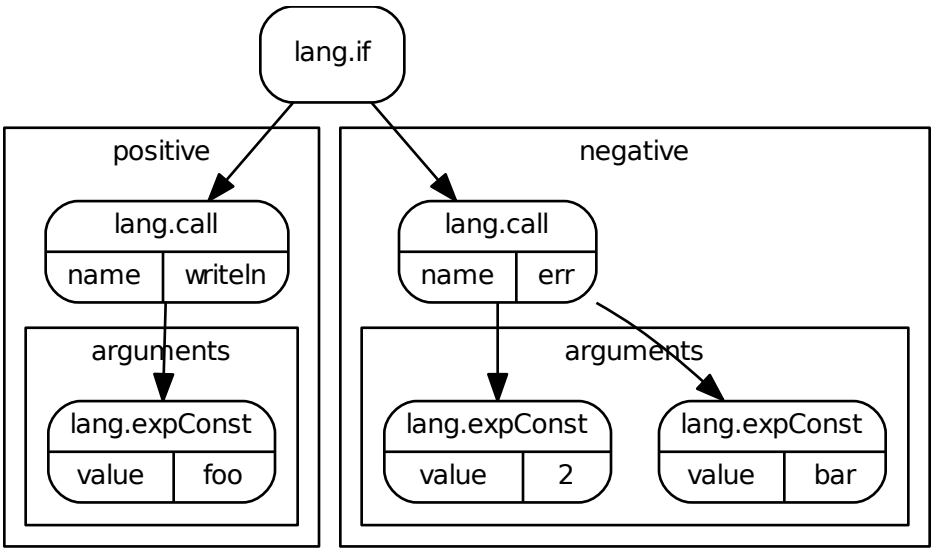
(continues on next page)

(continued from previous page)

```
else
  err(2, 'bar')
```

In a typical language (tm) the most probable error would be something like "Invalid predicate: expression `writeln('foo')` is not of type `boolean`" and "Missing keyword `then`". But the chosen indentation somehow hints that using the call to *writeln* as a predicate was not what the author intended to do.

In BlattWerkzeug the predicate may be omitted without touching the `positive` or `negative` branch. It is therefore trivial to tell the user that he has forgotten to supply a predicate.



3.3 Grammar Validation

Todo: Give an example for the `if`-statement that has been used above.

BlattWerkzeug uses its own grammar language that is very losely inspired by the RelaxNG compact notation. The mental model however is very similar to typical grammars, but is strictly concerned with the structure of the syntaxtree. A BlattWerkzeug grammar consists of a name and multiple node definitions:

```
grammar "ex1" {
  node "element" {
  }
}
```

This grammer defines a language named `ex1` which allows a single node with the name `element` to be present in the syntax tree. Allowed properties of nodes are defined as follows:

```
grammar "ex2" {
  node "element" {
    prop "name" { string }
  }
}
```

The curly brackets for the property need to denote at least the type of the property, valid values are `string` and `number`. Both of these properties may be limited further, see the section *Property Restrictions* for more details.

Multiple node definitions can be simply stated on after another as part of the `grammar` section:

```
grammar "ex3" {
  node "element" {
    prop "name" { string }
  }
  node "attribute" {
    prop "name" { string }
    prop "value" { string }
  }
}
```

Valid children of a node are defined via the `children` directive, a name and the corresponding "production rule". The production rule allows to specify sequences (using a space), alternatives (using a pipe "|") and "interleaving" (using the ampersand "&"). The mentioned elements can be quantified using the standard $\star$ (0 to unlimited), + (1 to unlimited) and ? (0 or 1) multiplicity operators. This example technically defines two sequences "elements" and "attributes" that allow zero or more occurences of the respective entity:

```
grammar "ex4" {
  node "element" {
    prop "name" { string }
    children "elements" ::= element*
    children "attributes" ::= attribute*
  }
  node "attribute" {
    prop "name" { string }
    prop "value" { string }
  }
}
```

3.3.1 Property Restrictions

3.3.2 Children Restrictions

3.3.3 Example Grammar: XML

```
grammar "dxml" {
  node "if" {
    children allowed "condition" ::= expr
    children allowed "body" ::= element* & text* & interpolate* & if*
  }
  typedef "expr" ::= exprVar | exprConst | exprBinary
  node "text" {
    prop "value" { string }
  }
```

(continues on next page)

(continued from previous page)

```
  node "element" {
    terminal "tag-open-begin" "<"
    prop "name" { string }
    children allowed "attributes" ::= attribute*
    terminal "tag-open-end" ">"
    children allowed "elements" ::= element* & text* & interpolate* & if*
    terminal "tag-close" "<ende/>"
  }
  node "exprVar" {
    prop "name" { string }
  }
  node "attribute" {
    prop "name" { string }
    terminal "equals" "="
    terminal "quot-begin" "\""
    children allowed "value" ::= text* & interpolate*
    terminal "quot-end" "\""
  }
  node "exprConst" {
    prop "name" { string }
  }
  node "exprBinary" {
    children allowed "lhs" ::= expr
    children allowed "operator" ::= binaryOperator
    children allowed "rhs" ::= expr
  }
  node "interpolate" {
    children allowed "expr" ::= expr
  }
  node "binaryOperator" {
    prop "operator" { string }
  }
}
```

3.3.4 Example Grammar: SQL

```
grammar "sql" {
  node "from" {
    terminal "keyword" "FROM"
    children sequence "tables", between: "," ::= tableIntroduction+
    children sequence "joins" ::= join*
  }
  typedef "join" ::= crossJoin | innerJoinUsing | innerJoinOn
  typedef "query" ::= querySelect | queryDelete
  node "where" {
    terminal "keyword" "WHERE"
    children sequence "expressions" ::= sql.expression sql.whereAdditional*
  }
  node "delete" {
    terminal "keyword" "DELETE"
  }
  node "select" {
    terminal "keyword" "SELECT"
    prop? "distinct" { boolean }
    children allowed "columns", between: "," ::= expression* & starOperator?
```

(continues on next page)

(continued from previous page)

```
  }
  node "groupBy" {
    terminal "keyword" "GROUP BY"
    children allowed "expressions", between: "," ::= sql.expression+
  }
  node "orderBy" {
    terminal "keyword" "ORDER BY"
    children allowed "expressions", between: "," ::= sql.expression+
  }
  node "constant" {
    prop "value" { string }
  }
  node "crossJoin" {
    children sequence "table" ::= tableIntroduction
  }
  node "parameter" {
    terminal "colon" ":"
    prop "name" { string }
  }
  node "columnName" {
    prop "refTableName" { string }
    terminal "dot" "."
    prop "columnName" { string }
  }
  typedef "expression" ::= columnName | binaryExpression | constant | parameter |␣
↪functionCall
  node "innerJoinOn" {
    terminal "keyword" "INNER JOIN"
    children sequence "table" ::= tableIntroduction
    terminal "keywordOn" "ON"
    children sequence "on" ::= expression
  }
  node "queryDelete" {
    children sequence "delete" ::= delete
    children sequence "from" ::= from
    children sequence "where" ::= where?
  }
  node "querySelect" {
    children sequence "select" ::= select
    children sequence "from" ::= from
    children sequence "where" ::= where?
    children sequence "groupBy" ::= groupBy?
    children sequence "orderBy" ::= orderBy?
  }
  node "functionCall" {
    prop "name" { string }
    terminal "paren-open" "("
    children sequence "arguments", between: "," ::= expression*
    terminal "paren-close" ")"
  }
  node "starOperator" {
    terminal "star" "*"
  }
  node "innerJoinUsing" {
    terminal "keyword" "INNER JOIN"
    children sequence "table" ::= tableIntroduction
    terminal "keywordUsing" "USING"
```

(continues on next page)

(continued from previous page)

```
    children sequence "using" ::= expression
  }
  node "whereAdditional" {
    prop "operator" { string { enum "and" "or" } }
    children sequence "expression" ::= expression
  }
  node "binaryExpression" {
    children sequence "lhs" ::= expression
    children sequence "operator" ::= relationalOperator
    children sequence "rhs" ::= expression
  }
  node "tableIntroduction" {
    prop "name" { string }
    prop? "alias" { string }
  }
  node "relationalOperator" {
    prop "operator" { string { enum "<" "<=" "=" ">=" ">" "LIKE" "NOT LIKE" } }
  }
}
```

CHAPTER 4

Block Languages

Block Languages are built with specific grammars in mind and can be thought of as an "representation layer" for syntaxtrees. Whilst the task of a grammar is to describe the structure of a tree, the task of a block language is to describe the visual representation. It does this via its own meta-description that may be either generated and maintained by hand or automatically generated from a grammar and some generation instructions.

This section of the manual initially describes the widgets that are available to represent a specific syntax tree. It then continues to describe how meaningful block languages may be automatically generated from a grammar.

4.1 Available widgets

The formal definition of the available widgets is part of the Typescript-namespace `VisualBlockDescriptions`. All available widgets are defined as a `JSON`-object that must at least define the `blockType`.

Additionally every widget may be styled using arbitrary `DOM`-properties using the optional `style`-object. The given properties will be applied directly to the given `DOM`-nodes, there is currently no support for "real" `CSS`.

4.1.1 Constant

Displays a constant value in the editor. This is meant to be used for keywords or keyword-like delimiters that may never be edited by the user. Constants are routinely meant to be styled with regards to their text colour and similar textual properties.

```
{
  "blockType": "constant",
  "text": "FROM",
  "style": {
    "color": "#0000ff",
    "width": "9ch",
    "display": "inline-block"
  }
}
```

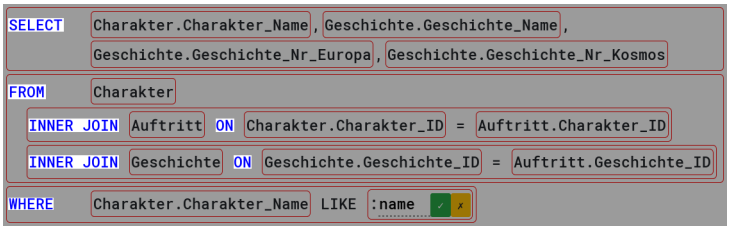


Fig. 1: Constants in the SQL language

4.1.2 Interpolated Values

Displays a property of a node that is represented in its block form. Although the result looks pretty much like a constant text on the outside, the value that is actually displayed will be determined depending on the syntaxtree that is loaded.

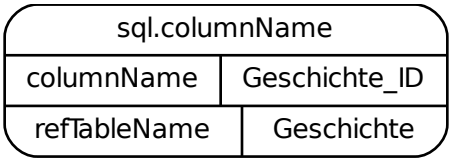


Fig. 2: Tree to visualize

```
{
  "blockType": "interpolated",
  "property": "columnName"
}
```

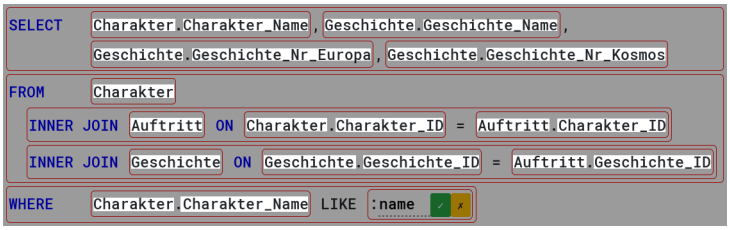


Fig. 3: Interpolated values in the SQL language

4.1.3 Editable Values

This property works almost like an interpolated value: It displays the value of a property. But if the user clicks the value an editor is opened.

```
{
  "blockType": "input",
  "property": "columName"
}
```

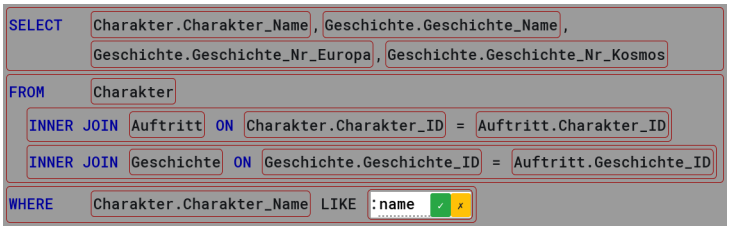


Fig. 4: Editing a value

4.1.4 Blocks

The actual blocks have no default appearance of their own but they may define child widgets that should be rendered. These blocks usually correspond to distinct types in grammars.

4.1.5 Iterating over children

So far every presented widget worked on atomic properties of a node. Child groups are rendered using an iterator which specifies the name of the child group to render.

4.2 Block Language Generation

Although it is possible to create block languages by hand, this approach does not scale too nicely with the idea of dynamically restricted programming environments. It would mean that e.g. different variants of `SQL` (that all share a common grammar) would require loads of duplicated effort to maintain.

CHAPTER 5

Project Structure

Fig. 1: High-level overview about different client and server components.

At its core the project consists of two codebases:

- A Ruby-server that uses Rails (found under `server`).
- A single page browser application that uses Typescript and Angular (found under `client`). This source code can be compiled in three different variants: `browser`, `universal` and `ide-service`.

But looking into it with more detail, the participating systems are responsible for the following tasks:

IDE Application (Browser) The "typical" browser application. This is the code that handles the actual user interaction like drag & drop events.

IDE Application (Universal) This variant of the application is basically a `node.js`-compatible version of the browser application which can be run on the server. This is used to speed up initial page loading and to be at least a little bit SEO-friendly. And apart from that it allows the non-IDE pages to be usable without JavaScript enabled.

IDE Service A commandline application that reads `JSON`-messages from `stdin` and outputs matching `JSON`-responses. This service ensures that server and client can run the exact same validation and compile operations without the need for any network roundtrips.

Webserver Although it is possible to run a server instance without dedicated webserver, this is strongly discouraged. The webserver should serve static files (like the compiled client), and route requests to the "Universal" application and the API-Server.

Rails API-Server This server acts as the storage backend: Clients store and retrieve the data of their projects via a `REST`-endpoints. The actual data is stored in the database and the filesystem and separated into three different environments: `production`, `development` and `test`, no data is shared between these environments. Additionally the server carries out operations that can't be done on the client, mainly database operations.

PostgreSQL Server The database stores most of the project data, basically everything besides from file assets like images and `.sqlite`-databases.

Project Data Folders Actual files, like images and `.sqlite`-databases are stored in the filesystem.

CHAPTER 6

Compilation Guide

This part of the documentation is aimed at people want to compile the project. As there are currently no pre-compiled distributions available, it is also relevant for administrators wanting to run their own server.

Note: Currently it is assumed that this project will built on a UNIX-like environment. Although building it on Windows should be possible, all helper scripts (and Makefiles) make a lot of *UNIX*-centric assumptions.

There are loads of fancy task-runners out there, but a "normal" interface to all programming-related tasks is still a `Makefile`. Most tasks you will need to do regulary are therefore available via `make`. Apart from that there are folders for schemas, documentation, example projects and helper scripts.

6.1 Environment Dependencies

At its core the server is a "typical" Ruby on Rails application which relies on the following software:

- Ruby >= 2.2.2 (Because of Rails 5.1)
- Postgres >= 9.3 (Because of JSON support)
- ImageMagick 6.9.9.34 (Version 7 is not yet properly supported by our used Ruby Library)
- FileMagick 5.32
- GraphViz 2.40.1
- SQLite >= 3.15.0 (with Perl compatible regular expressions)

Compiling the client requires the following dependencies (taken from here):

- NodeJS >= 6.9.0
- npm >= 3.0.0

Alternatively you may use Docker to run the server and compile the client.

6.1.1 Ubuntu Packages

Execute this command (works with Ubuntu 18.04):

```
sudo apt install ruby ruby-bundler ruby-dev postgresql-10 libpq-dev \
imagemagick libmagickcore-dev libmagickwand-dev \
magic libmagic-dev graphviz sqlite libsqlite3-dev \
nodejs npm
```

6.1.2 SQLite and PCRE

Database schemas created with BlattWerkzeug make use of regular expressions which are usually not compiled into the `sqlite3` binary. To work around this most distributions provide some kind of `sqlite3-pcre`-package which provides the regex implementation.

- Ubuntu: sqlite3-pcre
- Arch Linux: sqlite-pcre-git (AUR)

These packages should install a single library at `/usr/lib/sqlite3/pcre.so` which can be loaded with `.load /usr/lib/sqlite3/pcre.so` from the `sqlite3`-REPL. If you wish, you can write the same line into a file at `~/.sqliterc` which will be executed by `sqlite3` on startup.

6.1.3 DNS and Subdomains

BlattWerkzeug makes use of subdomains to render the public representation of a project. The development environment assumes, that any subdomains of `localhost.localdomain` will be routed to the `localhost`. The URL `http://cyoa.localhost.localdomain` should for example resolve to your `localhost` and would display the rendered index-page of the project `cyoa`. This works out of the box on various GNU/Linux-distributions, but as this behaviour is not standardised it should not be relied upon. To reliably resolve project-subdomains you should either write custom entries for each project in `/etc/hosts` or use a lightweight local DNS-server like Dnsmasq.

In a production environment you should run the server on a dedicated domain and route all subdomains to the same server instance.

6.1.4 PostgreSQL

The actual project code is stored in a PostgreSQL database. You will need to provide a user who is able to create databases.

6.2 Compiling and Running

Clone the sources from the git repository at BitBucket.

6.2.1 Running locally

- Ensure you have the "main" dependencies installed (`ruby` and `bundle` for the Server, `node` and `npm` for the client).

1. Compiling all variants of the client requires can be done by navigating to the `client` folder and executing the following steps.

1. `make install-deps` will pull all further dependencies that are managed by the respective packet managers. If this fails check that your environment meets the requirements: *Environment Dependencies*.
2. After that, the web application need to be compiled and packaged once: `make client-compile` for a fully optimized version or `make client-compile-dev` for a development version.
3. The server requires the special "IDE Service" variant of the client to function correctly. It can be created via `make cli-compile`.

2. Running the server requires the following steps in the `server` folder:
   (a) `make install-deps` will pull all further dependencies that are managed by the respective packet managers. If this fails check that your environment meets the requirements: *Environment Dependencies*.
   (b) Start a PostgreSQL-server that has a user `esqulino` who is allowed to create databases. You can alternatively specify your own user (see *database.yml at server/conf*).
   (c) Setup the database (*make setup-database*). This will create all required tables.
   (d) You may now run the server, to do this locally simply use `make run-dev` and it will spin up a local server instance listening on port `9292`. You can alternatively run a production server using `make run`.
3. You then need to seed the initial data that is part of this instance using `make load-all-data`. This will setup a pre-configured environment with some programming languages, block languages and projects.

The setup above is helpful to get the whole project running once, but if you want do develop it any further you are better of with the following options:

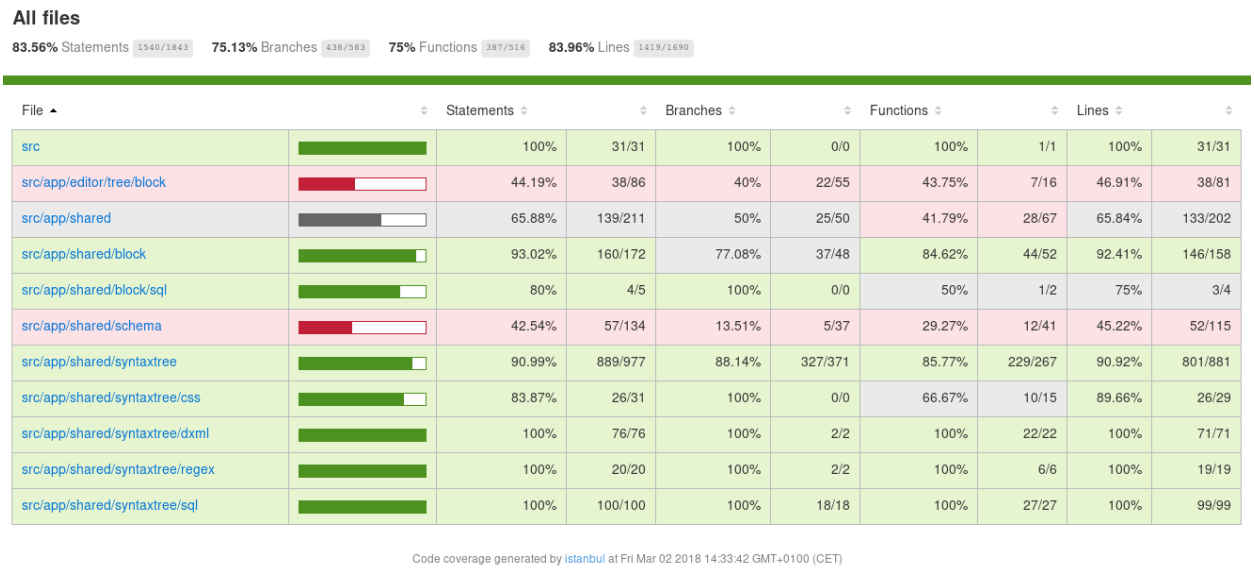
- Relevant targets in the `client` folder: * Run `NG_OPTS="--watch" make client-compile-dev` in the `client` folder. The `--watch` option starts a filesystem watcher that rebuilds the client incrementally on any change, which drastically reduces subsequent compile times. * Run `make client-test-watch` to continously run the client testcases in the background.
- Relevant targets in the `server` folder: * Run `make test-watch` to continously run the server testcases in the background. This requires a running PostgreSQL database server.

6.2.2 Testing and code coverage

Calling `make test` in the `client` folder will run the tests once against a headless version of Google Chrome and Firefox.

- `make test-watch` will run the tests continously after every change to the clients code.
- The environment variable `TEST_BROWSERS` controls which browsers will run the test, multiple browsers may be specified using a `,` and spaces are not allowed. The following values should be valid:
  - `Firefox` and `Chrome` for the non-headless variants that open dedicated browser windows.
  - `FirefoxHeadless` and `ChromeHeadless` that run in the background without any visible window.

After running tests the folder `coverage` will contain a navigateable code coverage report:



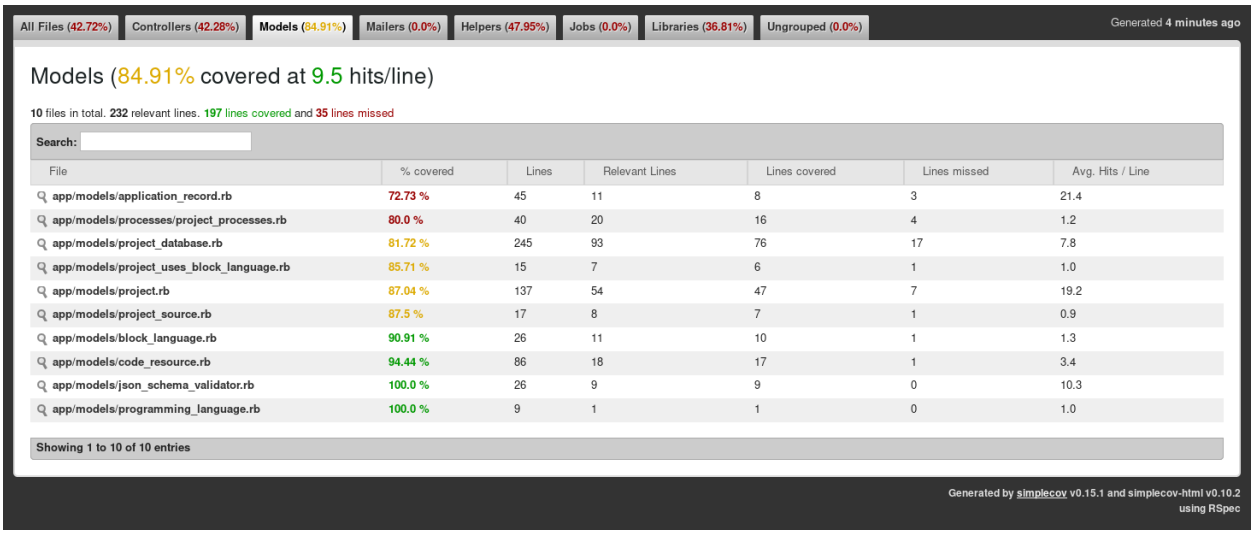
All files

83.56% Statements 1540/1843 75.13% Branches 438/583 75% Functions 387/516 83.96% Lines 1419/1690

File	Statements		Branches		Functions		Lines	
src	100%	31/31	100%	0/0	100%	1/1	100%	31/31
src/app/editor/tree/block	44.19%	38/86	40%	22/55	43.75%	7/16	46.91%	38/81
src/app/shared	65.88%	139/211	50%	25/50	41.79%	28/67	65.84%	133/202
src/app/shared/block	93.02%	160/172	77.08%	37/48	84.62%	44/52	92.41%	146/158
src/app/shared/block/sql	80%	4/5	100%	0/0	50%	1/2	75%	3/4
src/app/shared/schema	42.54%	57/134	13.51%	5/37	29.27%	12/41	45.22%	52/115
src/app/shared/syntaxtree	90.99%	889/977	88.14%	327/371	85.77%	229/267	90.92%	801/881
src/app/shared/syntaxtree/css	83.87%	26/31	100%	0/0	66.67%	10/15	89.66%	26/29
src/app/shared/syntaxtree/dxml	100%	76/76	100%	2/2	100%	22/22	100%	71/71
src/app/shared/syntaxtree/regex	100%	20/20	100%	2/2	100%	6/6	100%	19/19
src/app/shared/syntaxtree/sql	100%	100/100	100%	18/18	100%	27/27	100%	99/99

Code coverage generated by istanbul at Fri Mar 02 2018 14:33:42 GMT+0100 (CET)

Tests for the server are run in the same fashion: Call `make test` in the `server` folder to run them once, `make test-watch` run them continously. And again the folder `coverage` will contain a code coverage report:



All Files (42.72%) | Controllers (42.28%) | Models (84.91%) | Mailers (0.0%) | Helpers (47.95%) | Jobs (0.0%) | Libraries (36.81%) | Ungrouped (0.0%)

Generated 4 minutes ago

Models (84.91% covered at 9.5 hits/line)

10 files in total. 232 relevant lines. 197 lines covered and 35 lines missed

Search:

File	% covered	Lines	Relevant Lines	Lines covered	Lines missed	Avg. Hits / Line
app/models/application_record.rb	72.73 %	45	11	8	3	21.4
app/models/processes/project_processes.rb	80.0 %	40	20	16	4	1.2
app/models/project_database.rb	81.72 %	245	93	76	17	7.8
app/models/project_uses_block_language.rb	85.71 %	15	7	6	1	1.0
app/models/project.rb	87.04 %	137	54	47	7	19.2
app/models/project_source.rb	87.5 %	17	8	7	1	0.9
app/models/block_language.rb	90.91 %	26	11	10	1	1.3
app/models/code_resource.rb	94.44 %	86	18	17	1	3.4
app/models/json_schema_validator.rb	100.0 %	26	9	9	0	10.3
app/models/programming_language.rb	100.0 %	9	1	1	0	1.0

Showing 1 to 10 of 10 entries

Generated by simplecov v0.15.1 and simplecov-html v0.10.2 using RSpec

6.2.3 Running via Docker

There are pre-built docker images for development use on docker hub: marcusriemer/sqlino. These are built using the various `Dockerfiles` in this repository and can also be used with the `docker-compose.yml` file which is also part of this repository. Under the hood these containers use the same `Makefiles` and commands that have been mentioned above.

Depending on your local configuration you might need to run the mentioned `Makefile` with `sudo`.

- `make -f Makefile.docker pull-all` retrieves the most recent version of all images from the docker hub.
- `make -f Makefile.docker run-dev` starts docker containers that continously watch for changes to the `server` and `client` folders. It mounts the projects root folder as volumes into the containers, which allows you to edit the files in `server` and `client` in your usual environment. A third container is started for PostgreSQL.

- `make -f Makefile.docker shell-server-dev` opens a shell inside the docker container of the server. You might require this to do maintenance tasks with `bin/rails` for the server.

CHAPTER 7

Configuration Guide

This part of the documentation is aimed at people want to run the project. It assumes familarity with Linux and typical server software like databases and webservers.

7.1 Environments and Settings

All settings may be configured per environment (`PRODUCTION`, `DEVELOPMENT`, `TEST`). The most important options can all be found in the `sqlino.yml`

7.1.1 Storage

The server currently uses two places to store its data:

- The data folder may be configured via the `data_dir` key in `server/conf/sqlino.yml`.
- The database is configured via Rails in `server/conf/database.yml`

Additionaly the expects to find certain assets in configurable locations (`sqlino.yml`):

- `client_dir` must point to the compiled client with files like `index.html` and different `*.bundle.js` files.
- `schema_dir` must point to a folder that contains various `*.json`-schema files.

7.2 Server side rendering

You may initially render pages on the server. This drastically speeds up initial load times and provides a partial fallback for users that disable JavaScript.

7.3 Backing up and seeding data

The `server/Makefile` contains two targets that allow to im- or export data to a running server instance: `load-all-data` and `dump-all-data`. The system is *very* basic at the moment and not formally tested, for proper backup purposes.

That said, the following things need to be included in a backup for any environment:

- The Postgres-database as denoted in `server/config/database.yml`
- The `data_dir` as denoted in `server/config/sqlino.yml`

7.4 Example configuration files

This section contains some exemplary configuration files that work well for the official server at blattwerkzeug.de.

7.4.1 `sqlino.yml` at `server/conf`

```
development:
  name: "Blattwerkzeug (Dev)"
  data_dir: ../data/dev
  client_dir: ../client/dist/browser
  schema_dir: ../schema/json
  project_domains: ["localhost.localdomain:9292"]
  editor_domain: "localhost.localdomain:9292"
  mail:
    default_sender: "BlattWerkzeug <blattwerkzeug@gurxite.de>"
    admin: "Marcus@GurXite.de"
  ide_service:
    exec:
      node_binary: /usr/bin/node
      program: ../client/dist/cli/main.cli.js
      mode: one-shot
  seed:
    data_dir: ../seed

test:
  name: "Blattwerkzeug (Test)"
  data_dir: ../data/test
  client_dir: ../client/dist/browser
  schema_dir: ../schema/json
  project_domains: ["localhost.localdomain:9292"]
  editor_domain: "localhost.localdomain:9292"
  mail:
    default_sender: "BlattWerkzeug <blattwerkzeug@gurxite.de>"
    admin: "Marcus@GurXite.de"
  # The IDE service will, under most circumstances, honor the
  # "mock: true" setting. This allows testcases to specify arbitrary
  # languages (and speeds up the whole ordeal).
  # But some tests verify that the actual code runs correctly,
  # so the "exec" configuration here may not be removed.
  ide_service:
    mock: true
    exec:
```

(continues on next page)

(continued from previous page)

```
      node_binary: /usr/bin/node
      program: ../client/dist/cli/main.cli.js
      mode: one-shot
  seed:
    data_dir: ../seed

production:
  name: "Blattwerkzeug"
  data_dir: ../data/dev
  client_dir: ../client/dist/browser
  schema_dir: ../schema/json
  project_domains: ["blattzeug.de"]
  editor_domain: "blattwerkzeug.de"
  mail:
    default_sender: "BlattWerkzeug <blattwerkzeug@gurxite.de>"
    admin: "Marcus@GurXite.de"
  ide_service:
    exec:
      node_binary: /usr/bin/node
      program: ../client/dist/cli/main.cli.js
      mode: one-shot
  seed:
    data_dir: ../seed
```

7.4.2 `database.yml` at `server/conf`

```
default: &default
  adapter: postgresql
  database: esqulino
  host: <%= ENV['DATABASE_HOST'] || 'localhost' %>
  username: esqulino
  encoding: unicode

development:
  <<: *default
  database: esqulino_dev

test:
  <<: *default
  database: esqulino_test

production:
  <<: *default
  database: esqulino_prod
```

7.4.3 Example `systemd` configuration

Listing 1: blattwerkzeug.service

```
[Unit]
Description=BlattWerkzeug - Backend Server
After=network.target
```

(continues on next page)

(continued from previous page)

```
# CUSTOMIZE: Optionally add `blattwerkzeug-universal` as a dependency
Wants=postgresql nginx

[Service]
# CUSTOMIZE: Generate a secret using `rake secret`
Environment="SECRET_KEY_BASE=<CUSTOMIZE ME>"
# CUSTOMIZE: Use a dedicated user
User=blattwerkzeug
# CUSTOMIZE: Set the correct path
WorkingDirectory=/srv/htdocs/blattwerkzeug/
ExecStart=/usr/bin/make -C server run

[Install]
WantedBy=multi-user.target
```

Listing 2: blattwerkzeug-universal.service

```
[Unit]
Description=BlattWerkzeug - Universal Rendering Server
After=network.target

[Service]
# CUSTOMIZE: Use a dedicated user
User=blattwerkzeug
# CUSTOMIZE: Set the correct path
WorkingDirectory=/srv/htdocs/blattwerkzeug/
ExecStart=/usr/bin/make -C client universal-run

[Install]
WantedBy=multi-user.target
```

7.4.4 Example `nginx` configuration

```
# The main IDE server
server {
    listen 80;
    listen 443 ssl http2;

    # CUSTOMIZE: Add ssl certificates

    # CUSTOMIZE: Change domains and paths
    server_name www.blattwerkzeug.de blattwerkzeug.de;
    root /srv/htdocs/esqulino.marcusriemer.de/client/dist/browser;
    error_log /var/log/nginx/blattwerkzeug.de-error.log error;
    access_log /var/log/nginx/blattwerkzeug.de-access.log;

    index index.html;

    # The most important route: Everything that has the smell of the API
    # on it goes to the API server
    location /api/ {
        proxy_pass http://127.0.0.1:9292;
        proxy_set_header Host $host;
```

(continues on next page)

(continued from previous page)

```
        add_header 'Access-Control-Allow-Origin' '*';
        add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS';
        add_header 'Access-Control-Allow-Headers' 'DNT,X-CustomHeader,Keep-Alive,User-
↪Agent,X-Requested-With,If-Modified-Since,Cache-Control,Content-Type';
    }

    # Static assets should be served by nginx, no matter what
    location ~* \.(css|js|svg|png)$ {
        gzip_static on;
    }

    # Attempting to hand off requests to the universal rendering
    # server, but fail gracefully if no universal rendering is available
    location @non_universal_fallback {
        try_files $uri /index.html;
        gzip_static on;
        break;
    }

    location ~ ^(/$|/about) {
        error_page 502 = @non_universal_fallback;

        proxy_pass http://127.0.0.1:9291;
        proxy_set_header Host $host;
        proxy_intercept_errors  on;
    }

    # Everything that ends up here is served by the normal filesystem
    location / {
        try_files $uri /index.html;
        gzip_static on;
    }
}

# Rendering projects on subdomains
server {
    listen 80;
    listen 443 ssl http2;

    # CUSTOMIZE: Change domains and paths
    server_name *.blattwerkzeug.de *.blattzeug.de;
    error_log /var/log/nginx/blattwerkzeug.de-error.log error;
    access_log /var/log/nginx/blattwerkzeug.de-access.log;

    location / {
        proxy_pass http://127.0.0.1:9292;
        proxy_set_header Host $host;
    }

}
```

CHAPTER 8

The Online Platform

One of the main reasons for the development of BlattWerkzeug was the barrier of entry when a pupil attempts to make her or his first steps. Databases need to be obtained, possibly configured, servers need to be installed and maintained ... The web-based nature of BlattWerkzeug simplifies this process to basicly "surf to this page and get going". But offering BlattWerkzeug as a web application comes with additional benefits other than simplicity: The code is available from every computer and pupils can trivially share the results of their work with the rest of the world.

But these benefits do come with a prize: In order to work properly BlattWerkzeug needs to implement and maintain a lot of things that are very atypical for an IDE. The most prominent requirement is a robust way to separate data from different users. This can obviously be solved via some kind of user registration & login process, which is more a design than a technical challenge.

8.1 Types of Users

The masters thesis of Marcus Riemer describes a very rough outline of possible user groups in chapter 3.4.3. These considerations are however strictly limited to projects, as the online platform was considered to be out of scope for the masters thesis.

Without getting into details considering rights management we expect every registered user to take at least one of the following roles:

Learners want to create stuff using the IDE. They are busy creating new content that they want to demonstrate to their peers (or they actually want to learn something for the sake of learning, which would also be nice)[1].

Educators want to demonstrate programming concepts using the IDE. They are busy creating new content that they want to show to learners so those can adapt and remix them.

Moderators are required to ensure that the platform is not abused. They ensure that e.g. no copyrighted or pornographic content is shared via the platform.

With these roles in mind we can take a look at the different groups of people that make up the target audience:

[1] Note to self: Is there a distinction between "creators" and "learners" in established didactic concepts?

Teachers in a classroom setting are immediately responsible for a set of pupils. These students are very likely to be tasked with the same assignment so it needs to be as easy as possible to "share" the same project to multiple users.

Pupils in a classroom setting are supervised by a teacher and are expected to fulfill predetermined tasks.

Independent Creators want to create programs that are actually useful for some kind of problem that they have.

Independent Educators want to share their knowledge.

Visitors are not interested in the IDE itself, but in the things that have been created using the IDE.

8.2 Inspiration

With the rise of decentralized version control systems like Git and Mercurial came quite a few online platforms that offer some mixture of repository hosting and "social" features that ease collaboration (GitHub, BitBucket, GitLab, ...). As BlattWerkzeug strives to be a learning environment it should be as easy as possible (and encouraged) to learn from other peoples code.

The following questions may be helpful when thinking about the community and online platform aspects:

- How does the user registration process work?
  - Classic email registration seems to be a must, but what about different providers (school accounts, social media accounts, ...)?
  - Should a teacher be able to sign up (and manage?) his students?
  - Is the registration process different depending on the role?
  - How is the registration process secured against bots?
- What information should a user page contain?
  - How can a user give a spotlight to her or his most relevant projects?
  - Should there be different user pages depending on the role?
  - How (should?) a user show his expertise?
  - How (should?) a user link to his profile on other places?
- How or where do users communicate?
  - This does not only mean "social" communication but also feedback from teachers.
  - Should there be a possibility to comment on users, projects, databases, ... ?
  - Should there be any form of free-form discussion[2]?
  - Should there be any form of private discussion?
- How do users discover content that is relevant to them?
  - Some kind of tagging or categorization system?
  - Some kind of course system?
  - Some kind of referal system?
- How can projects be shared among multiple users?
  - Is "cloning" or "forking" a viable concept?

[2] Technical detail: Maybe an existing application like Discourse would be a good fit?

- Should certain resources be read-only in forked projects?

8.3 Technical Requirements

BlattWerkzeug technically consists of two different codebases: A Ruby on Rails application for the server and an Angular application for the client. See *Project Structure* for the general overview.

As the server uses the so called `API`-mode of Rails quite a few of the "standard" gems for user authentication won't work without some degree of customization. The model & controller functionality of gems like devise *may* be helpful, but due to the Angular Client there is no view rendering available. A standard like JSON Web Tokens (RFC 7519) seems like the most viable solution to bridge the gap between the ruby code on the server and the client.

CHAPTER 9

Storage Formats

9.1 Schema for the Abstract Syntax Tree

This description regulates how all ASTs should be stored when written to disk or sent over the wire. It requires every node to at least tell its name and some hint how a node can be constructed at runtime. The data of a node is split up in two broader categories: Children, which may be nested and properties, which should not allow any nesting.			
type	*object*		
properties			
• children	Nodes may have children in various categories. This base class makes no assumptions about the names of children. Examples for children in multiple categories would be things like “attributes” and generic “children” in a specialization for XML.		
	type	*object*	
	additionalProperties	type	*array*
		items	
		•	#/definitions/NodeDescription
• **language**	This is effectively a namespace, allowing identical names for nodes in different languages.		
	type	*string*	
• **name**	The name of this not, this is used to lookup the name of a corresponding type.		
	type	*string*	
• properties	Nodes may have all kinds of properties that are specific to their concrete use.		
	type	*object*	
	additionalProperties	type	*string*
additionalProperties	False		

CHAPTER 10

Glossary

Abstract Syntax Tree (AST) The AST is a tree representation of the syntactic structure of a document. It is analyzed by the validator to determine logical errors in the code and can be handed over to a compiler to receive a "normal" text representation of the tree in question.

Index

A