BlattWerkzeug Documentation

Release 0.1

Marcus Riemer

Apr 02, 2020

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 groups**. Each of these groups has a name and contains any number of subtrees. 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): ../schema/index.

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 `numRattles` that is mentioned.

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

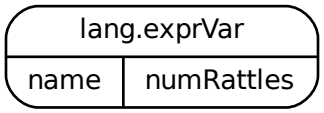


Fig. 2: Expression `numRattles`

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`. Each child group contains a sub-tree of its own and technically any node may have any number of disjoint subtrees.

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

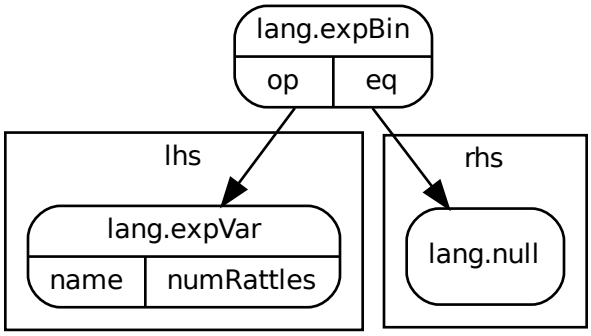


Fig. 3: Expression `numRattles == null`

CHAPTER 3

Grammars by Example

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 as the inspiration to describe the grammars in a user friendly textual representation.

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.

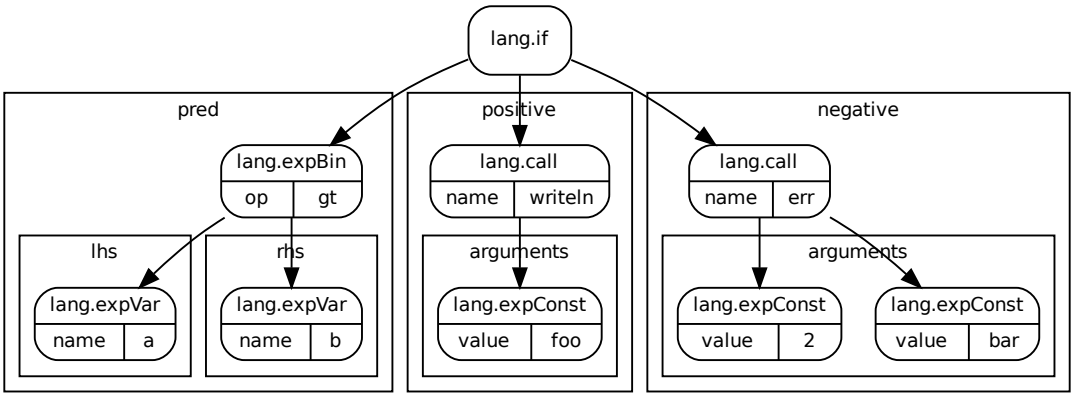
3.2.1 Example AST: `if`-Statement

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')
```

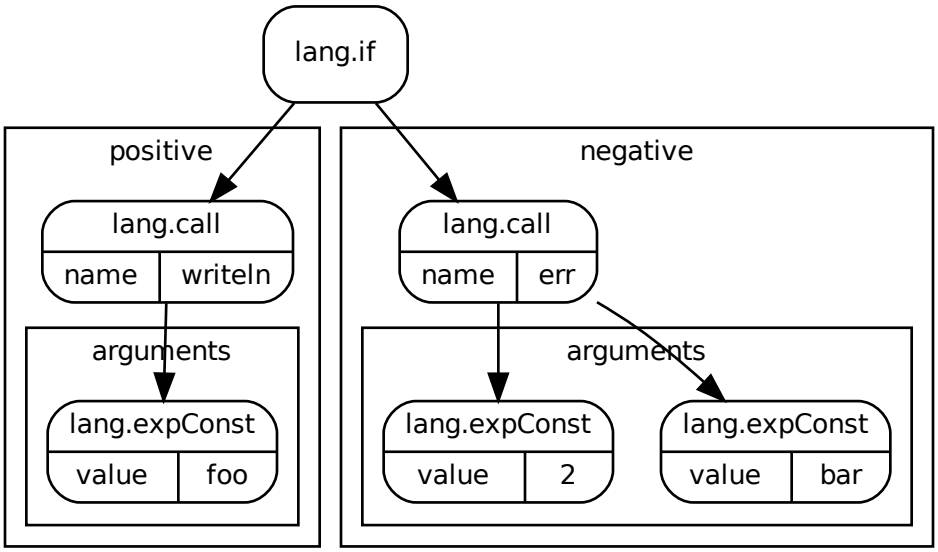
(continues on next page)

(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 Examples

The following chapters give various examples of language grammars that could be used in BlattWerkzeug. They are meant to serve as meaningful tutorials, not as a thorugh documentation.

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.

3.3.1 XML

In this example we will create a grammar that is able to describe `XML` like trees. Lets start with an almost empty grammar:

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

(continues on next page)

(continued from previous page)

```
  }
}
```

This grammer defines a language named `ex1` which allows a single node with the name `element` to be present in the syntax tree. This node may not have any children or properties, so the only valid syntaxtree would consist of a single node.

In order to allow nodes to be named, we introduce a `property`:

```
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 `boolean`, `string` and `number`. The latter of these properties may be limited further, see the section *Property Restrictions* for more details.

Multiple node definitions can be simply stated one 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 }
  }
}
```

CHAPTER 4

Grammar Reference

This section is the formal documentation of the grammar languages used by BlattWerkzeug. These grammars define the general validity of syntax trees and describe programming languages. This chapter assumes that you have read and understood the following other chapters:

- The chapter *The Abstract Syntax Tree* which describes the data structure that requires validation.
- The chapter *Grammars by Example* which gives an informal introduction and describes how the grammars that are described here differ from "typical" grammars as they are used in compiler construction.

4.1 Relationship to RelaxNG

The concepts of the grammar language are based on the concepts of XML validation as described by Relax NG. The key difference is, that `XML` always has exactly two types of children to consider: attributes and other elements. The grammar described in this document makes a similar distinction between so called `properties` and `children`. But it additionally allows multiple so called `child groups`, which are all individual sub-trees.

4.2 Top level type definitions

Every top level definition introduces a new type that can be referenced. Grammars and SyntaxTrees are linked via the `language` and `name` properties of a node. These properties combined form a fully qualified typename, which can be looked up in the grammar.

This lookup may yield one of two different type definitions: A definition of type `node` matches an actual node in an abstract syntax tree, it may define properties and children. A `typedef` on the other hand denotes a type that will never exist in a tree, it's a mere placeholder for a set of other types that could appear in the referenced position.

4.3 Top level: `node`

This type defines which attributes (properties or children) a certain node may have. Both types of attributes share a common namespace, it is therefore not possible to have a property **and** a child group named `foo` on the same `node` definition.

In the pretty printed form of a grammar, a node may be denoted as follows:

```
node "example."name" {
  # Attributes ("children" or "prop")
}
```

4.4 Property Restrictions

Properties are atomic values and may currently be `boolean`, `integer` or `string` values. The actual values are always stored as strings in the syntaxtree, they do **not** use the corresponding `JSON` types.

All properties may be defined as beeing `optional`, this is denoted by a `?` that follows the propertyname. If an optional property is absent from a syntaxtree during validation, this is not considered as an error.

4.4.1 `boolean`

Allows exactly the property values `true` and `false`. In the pretty printed form of a grammar, a node with a single boolean property may be denoted as follows:

```
node "example"."booleanProperty"  {
  prop "b" { boolean }
}
```

4.4.2 `integer`

Integers may specify restrictions of type `minInclusive` or type `maxInclusive`. In the pretty printed form of a grammar, a node with three diffrent ranges for integer properties may be denoted as follows:

```
node "example"."intgerProperty"  {
  prop "i" { integer }
  prop "fiveOrMore" { integer  5 }
  prop "fiveOrLess" { integer  5 }
  prop "zeroToFive" {
    integer {
       5
       0
    }
  }
}
```

4.4.3 `string`

Strings may be restricted in the following ways:

- It must have exactly (`length`), at least (`minLength`) or at most (`maxLength`) a certain length.

- It must be exactly one value of an enumeration (`enum`).
- It must match a regular expression (`regex`). This is the most flexible option (and it can be used to express all of the other restrictions), but it does lead to hard to understand error messages.

4.5 Children Restrictions

As every node in the syntaxtree may have any number of named subtrees, the grammar must be able to validate any number of subtrees for a certain type. Technically every childgroup may contain a list of subtrees in which every tree can be validated individually. Grammars may enforce rules about the order or cardinality for the types of the roots of those trees.

4.5.1 Childgroup Type `sequence`

Sequences expect an exact series of types in a certain child group. The following example shows a sequence where a valid syntax tree must have exactly four nodes overall:

```
node "sequence"."root" {
  children sequence "Children" ::= B A B
}
node "sequence"."B" { }
node "sequence"."A" { }
```

4.5.2 Childgroup Type `allowed`

For some kinds of subtrees the order of the following root nodes is irrelevant, but the cardinality may be very relevant. This is very common in markup languages, where many different types of children may be allowed in no particular order.

The following example defines the structure of some kind of document: It must have `Text`, it may have exactly a single `Figure` and it may contain any number of `Reference`:

```
node "allowed"."Document" {
  children allowed "Children" ::= Text+ & Figure? & Reference*
}
```

4.5.3 Limitation: No mixed groups

Note that it is currently **not** possible to mix e.g. `sequence` and `allowed` child groups as it would be possible with RelaxNG. This is mainly because no proper use case has surfaced that would warrant this rather complicated behavior. Under most circumstances using multiple child groups is a perfectly fine workaround. In order to add a single "Heading" for the `Document` type mentioned above, one could make the following workaround:

```
node "allowed"."Document" {
  children sequence "Heading" ::= Text
  children allowed "Body" ::= Text+ & Figure? & Reference*
}
```

Now every `Document` requires a single `Text` node in the `Heading` childgroup.

4.6 Top level: `typedef`

A `typedef` denotes a type that will never exist in a tree, it's a mere placeholder for a set of other types that could appear in the referenced position. This is useful when in certain places different but related types could be expected. Instead of repeating sets like `{unaryExpression, binaryExpression, constant}` again and again, a single typedef may group these common usage together.

Technically this doesn't add new functionality to the grammar language as a whole.

CHAPTER 5

Builtin Grammars

These grammars are currently shipped with the project.

5.1 SQL

Basic SQL as it is teached in most schools, no support for all sorts of imperative constructs.

```
grammar "sql" {
  typedef "sql"."query" ::= querySelect | queryDelete
  node "sql"."querySelect" {
    container vertical {
      children sequence "select" ::= select
      children sequence "from" ::= from
      children sequence "where" ::= where?
      children sequence "groupBy" ::= groupBy?
      children sequence "orderBy" ::= orderBy?
    }
  }
  node "sql"."select" {
    container horizontal {
      "SELECT"
      children sequence "distinct" ::= distinct?
      container horizontal {
        children allowed "columns" ::= (expression* & starOperator?)
      }
    }
  }
  node "sql"."distinct" {
    "DISTINCT"
  }
  node "sql"."from" {
    "FROM"
    children sequence "tables", between: "," ::= tableIntroduction+
```

(continues on next page)

(continued from previous page)

```
    container vertical {
      children sequence "joins" ::= join*
    }
  }
  node "sql"."tableIntroduction" {
    prop "name" { string }
  }
  typedef "sql"."join" ::= crossJoin | innerJoinUsing | innerJoinOn
  node "sql"."crossJoin" {
    children sequence "table" ::= tableIntroduction
  }
  node "sql"."innerJoinUsing" {
    "INNER JOIN"
    children sequence "table" ::= tableIntroduction
    "USING"
    children sequence "using" ::= expression
  }
  typedef "sql"."expression" ::= columnName | binaryExpression | constant | parameter␣
↪| functionCall | parentheses
  node "sql"."columnName" {
    prop "refTableName" { string }
    terminal "dot" "."
    prop "columnName" { string }
  }
  node "sql"."binaryExpression" {
    children sequence "lhs" ::= expression
    children sequence "operator" ::= relationalOperator
    children sequence "rhs" ::= expression
  }
  node "sql"."relationalOperator" {
    prop "operator" { string enum "<" "<=" "=" "<>" ">=" ">" "LIKE" "NOT LIKE" }
  }
  node "sql"."constant" {
    prop "value" { string }
  }
  node "sql"."parameter" {
    terminal "colon" ":"
    prop "name" { string }
  }
  node "sql"."functionCall" {
    prop "name" { string }
    terminal "paren-open" "("
    children sequence "distinct" ::= distinct?
    children sequence "arguments", between: "," ::= expression*
    terminal "paren-close" ")"
  }
  node "sql"."parentheses" {
    terminal "parenOpen" "("
    children sequence "expression" ::= expression
    terminal "parenClose" ")"
  }
  node "sql"."innerJoinOn" {
    "INNER JOIN"
    children sequence "table" ::= tableIntroduction
    "ON"
    children sequence "on" ::= expression
  }
```

(continues on next page)

(continued from previous page)

```
  node "sql"."where" {
    "WHERE"
    children sequence "expressions" ::= expression whereAdditional*
  }
  node "sql"."whereAdditional" {
    prop "operator" { string enum "AND" "OR" }
    children sequence "expression" ::= expression
  }
  node "sql"."groupBy" {
    "GROUP BY"
    children allowed "expressions", between: "," ::= expression+
  }
  node "sql"."orderBy" {
    "ORDER BY"
    children allowed "expressions" ::= (expression* & sortOrder*)+
  }
  node "sql"."queryDelete" {
    children sequence "delete" ::= delete
    children sequence "from" ::= from
    children sequence "where" ::= where?
  }
  node "sql"."delete" {
    "DELETE"
  }
  node "sql"."sortOrder" {
    children sequence "expression" ::= expression
    prop "order" { string enum "ASC" "DESC" }
  }
  node "sql"."starOperator" {
    "*"
  }
}
```

5.2 Trucklino Program

The “Truck Programming Language”, first described in the Bachelors Thesis of Sebastian Popp.

```
grammar "trucklino-program" {
  node "trucklino_program"."program" {
    container vertical {
      children sequence "procedures" ::= procedureDeclaration*
      children sequence "main" ::= statement*
    }
  }
  node "trucklino_program"."procedureDeclaration" {
    container horizontal {
      terminal "function" "function "
      prop "name" { string }
      terminal "parenOpen" "("
      children sequence "arguments" ::= procedureParameter*
      terminal "parenClose" ")"
      terminal "bodyOpen" " {"
    }
    container vertical {
```

(continues on next page)

(continued from previous page)

```
      children sequence "body" ::= statement*
    }
    container horizontal {
      terminal "bodyClose" "}"
    }
  }
  node "trucklino_program"."procedureParameter" {
    prop "name" { string }
  }
  typedef "trucklino_program"."statement" ::= procedureCall | if | loopFor | loopWhile
  node "trucklino_program"."procedureCall" {
    prop "name" { string }
    terminal "parenOpen" "("
    children sequence "arguments" ::= booleanExpression*
    terminal "parenClose" ")"
  }
  typedef "trucklino_program"."booleanExpression" ::= sensor | negateExpression |␣
↪booleanBinaryExpression | booleanConstant
  node "trucklino_program"."sensor" {
    prop "type" { string enum "lightIsRed" "lightIsGreen" "canGoStraight" "canTurnLeft
↪" "canTurnRight" "canLoad" "canUnload" "isOnTarget" "isSolved" }
  }
  node "trucklino_program"."negateExpression" {
    container horizontal {
      terminal "not" "(NOT "
      children sequence "expr" ::= booleanExpression
      terminal "close" ")"
    }
  }
  node "trucklino_program"."booleanBinaryExpression" {
    container horizontal {
      terminal "open" "("
      children sequence "lhs" ::= booleanExpression
      terminal "spaceBefore" " "
      children sequence "operator" ::= relationalOperator
      terminal "spaceAfter" " "
      children sequence "rhs" ::= booleanExpression
      terminal "close" ")"
    }
  }
  node "trucklino_program"."relationalOperator" {
    prop "operator" { string enum "AND" "OR" }
  }
  node "trucklino_program"."booleanConstant" {
    prop "value" { string enum "true" "false" }
  }
  node "trucklino_program"."if" {
    container horizontal {
      terminal "if" "if"
      terminal "parenOpen" " ("
      container horizontal {
        children sequence "pred" ::= booleanExpression
      }
      terminal "parenClose" ")"
      terminal "bodyOpen" " {"
    }
    container vertical {
```

(continues on next page)

(continued from previous page)

```
      children sequence "body" ::= statement*
    }
    container vertical {
      terminal "bodyClose" "}"
      children sequence "elseIf" ::= ifElseIf*
      children sequence "else" ::= ifElse?
    }
  }
  node "trucklino_program"."ifElseIf" {
    container horizontal {
      terminal "elseIf" "else if"
      terminal "parenOpen" " ("
      children sequence "pred" ::= booleanExpression
      terminal "parenClose" ")"
      terminal "bodyOpen" " {"
    }
    container vertical {
      children sequence "body" ::= statement*
      terminal "bodyClose" "}"
    }
  }
  node "trucklino_program"."ifElse" {
    container horizontal {
      terminal "else" "else"
      terminal "bodyOpen" " {"
    }
    container vertical {
      children sequence "body" ::= statement*
      terminal "bodyClose" "}"
    }
  }
  node "trucklino_program"."loopFor" {
    container horizontal {
      terminal "for" "for"
      terminal "parenOpen" " ("
      prop "times" { integer  0 }
      terminal "parenClose" ")"
      terminal "bodyOpen" " {"
    }
    children sequence "body" ::= statement*
    terminal "bodyClose" "}"
  }
  node "trucklino_program"."loopWhile" {
    container horizontal {
      terminal "while" "while"
      terminal "parenOpen" " ("
      children sequence "pred" ::= booleanExpression
      terminal "parenClose" ")"
      terminal "bodyOpen" " {"
    }
    container vertical {
      children sequence "body" ::= statement*
    }
    terminal "bodyClose" "}"
  }
}
```

5.3 JSON

This grammar is based upon the formal grammar definition found at json.org. It mimics the exact same structure but does not bother with whitespace.

```
grammar "json" {
  typedef "json"."value" ::= string | number | boolean | object | array | null
  node "json"."string" {
    terminal "quot-begin" "\""
    prop "value" { string }
    terminal "quot-end" "\""
  }
  node "json"."number" {
    prop "value" { integer }
  }
  node "json"."boolean" {
    prop "value" { boolean }
  }
  node "json"."object" {
    terminal "object-open" "{"
    children allowed "values", between: "," ::= key-value*
    terminal "object-close" "}"
  }
  node "json"."key-value" {
    children allowed "key" ::= string
    terminal "colon" ":"
    children allowed "value" ::= value
  }
  node "json"."array" {
    terminal "array-open" "["
    children allowed "values", between: "," ::= value*
    terminal "array-close" "]"
  }
  node "json"."null" {
    terminal "value" "null"
  }
}
```

5.4 CSS

Definition of CSS stylesheets, which are basicly a set of rules which in turn are defined by selectors and declarations.

```
grammar "css" {
  node "css"."document" {
    children allowed "rules" ::= rule+
  }
  node "css"."rule" {
    children sequence "selectors" ::= selector+
    terminal "rule-open" "{"
    children allowed "declarations" ::= declaration+
    terminal "rule-close" "}"
  }
  typedef "css"."selector" ::= selectorType | selectorClass | selectorId |␣
↪selectorUniversal
  node "css"."selectorType" {
```

(continues on next page)

(continued from previous page)

```
    prop "value" { string }
  }
  node "css"."selectorClass" {
    prop "value" { string }
  }
  node "css"."selectorId" {
    prop "value" { string }
  }
  node "css"."selectorUniversal" {
  }
  node "css"."declaration" {
    children choice "name" ::= propertyName
    terminal "colon" ":"
    children choice "value" ::= exprColor | exprAny
  }
  node "css"."propertyName" {
    prop "name" { string }
  }
  node "css"."exprColor" {
    prop "value" { string }
  }
  node "css"."exprAny" {
    prop "value" { string }
  }
}
```

5.5 Dynamic XML

```
grammar "dxml" {
  node "dxml"."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 "dxml"."attribute" {
    prop "name" { string }
    terminal "equals" "="
    terminal "quot-begin" "\""
    children allowed "value" ::= text* & interpolate*
    terminal "quot-end" "\""
  }
  node "dxml"."text" {
    prop "value" { string }
  }
  node "dxml"."interpolate" {
    children allowed "expr" ::= expr
  }
  typedef "dxml"."expr" ::= exprVar | exprConst | exprBinary
  node "dxml"."exprVar" {
    prop "name" { string }
  }
```

(continues on next page)

(continued from previous page)

```
  node "dxml"."exprConst" {
    prop "name" { string }
  }
  node "dxml"."exprBinary" {
    children allowed "lhs" ::= expr
    children allowed "operator" ::= binaryOperator
    children allowed "rhs" ::= expr
  }
  node "dxml"."binaryOperator" {
    prop "operator" { string }
  }
  node "dxml"."if" {
    children allowed "condition" ::= expr
    children allowed "body" ::= element* & text* & interpolate* & if*
  }
}
```

CHAPTER 6

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.

6.1 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`.

6.1.1 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"
  }
}
```

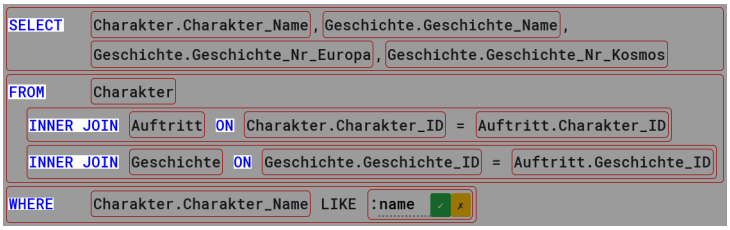


Fig. 1: Constants in the SQL language

6.1.2 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.

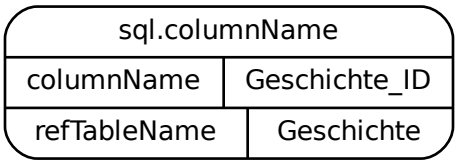


Fig. 2: Tree to visualize

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

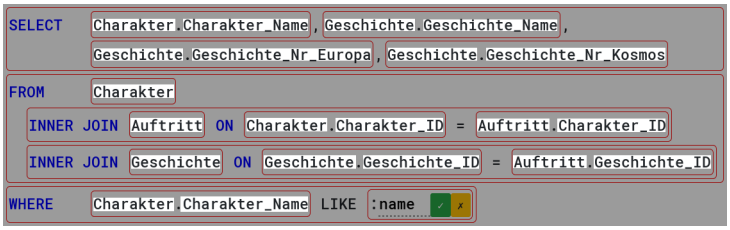


Fig. 3: Interpolated values in the SQL language

6.1.3 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"
}
```

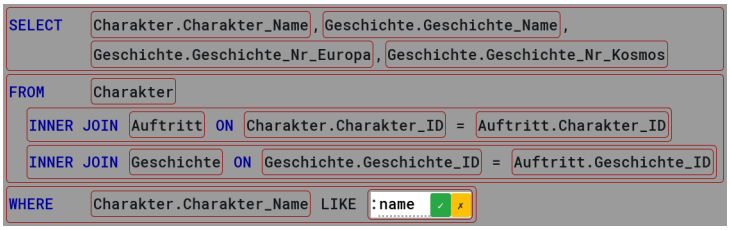


Fig. 4: Editing a value

6.1.4 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.

6.1.5 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.

6.2 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 7

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 8

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. If you are extending the project, please be sure to read the *Programming Guidelines*.

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.

8.1 Environment Dependencies

At its core the server is a "typical" Ruby on Rails application which relies on the following software. The given versions are a known konfiguration, more recent versions will probably work as well.

- Ruby >= 2.5.0 (Because of Rails 6.0)
- Postgres >= 9.4 (Because of JSON support)
- ImageMagick 7.0.8 (Version 6 should work as well)
- 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 >= 10.9.0
- npm >= 6.0.0

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

8.1.1 Ubuntu Packages

Execute this command to install the dependencies in a single step (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 sqlite3-pcre \
  nodejs npm
```

The versions of `nodejs` and `npm` on Ubuntu are sometimes badly outdated. In that case you probably want to use the binary distributions by NodeSource.

8.1.2 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.

8.1.3 DNS and Subdomains

BlattWerkzeug requires subdomains for two different purposes:

- The application itself is available in multiple languages. `en.blattwerkzeug.localdomain` should render the english version, `de.blattwerkzeug.localdomain` the german version.
- The web-projects will be rendered on their own subdomains.

The currently configured environment uses `lvh.me` to have a "proper" domain to talk to, which eases the initially required setup. It additionally allows to properly test the OAuth2 workflows with Google, which forbids the use of `localhost.localdomain` as a redirection target.

Alternatively you may configure the `localdomain` to be routed to the `localhost`. 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.

8.1.4 PostgreSQL

The actual project code is stored in a PostgreSQL database. You will need to provide a user who is able to create databases. For development you should stick to the default options that are provided in the `server/config/database.yml` file.

8.1.5 Environment Variables

The default environment assumes a readily available database which is configured via the `server/config/database.yml` file which happily picks up environment variables. As long as you are happy with those defaults, there is nothing to worry about. But some services do require customized information via environment variables.

- Various login providers that work via OAuth2 require a client ID and a client secret:
  - Google: `GOOGLE_CLIENT_ID`, `GOOGLE_CLIENT_SECRET`, these values are available from the Google Developer Console (if you are part of the BlattWerkzeug-project).
- Sending mails requires a configured `SMTP`-server: `SMTP_HOST`, `SMTP_USER`, `SMTP_PASS`

You probably want to use some tool like direnv (available for most Linux distros) to automatically manage these variables. Just install a hook to `direnv` in the `rc`-file of your shell and restart the shell. Then you can create a `.envrc` file in the server folder that contains something along the lines of:

```
export GOOGLE_CLIENT_ID=foo
export GOOGLE_CLIENT_SECRET=bar
```

Entering and leaving the folder will then automatically load and unload the mentioned environment variables.

8.2 Compiling and Running

Clone the sources from the git repository at BitBucket.

8.2.1 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:
   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. Start a PostgreSQL-server that has a user who is allowed to create databases.
   3. Setup the database and fill the database (`make reset-live-data`). This will create all required tables and load some sample data.
   4. 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`.
   5. If you require administrative rights, *you can give the permissions via the Rails shell*.

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 options descibed in *Loading and storing seed data*.

8.2.2 Running via Docker

There are pre-built docker images for development use on docker hub: marcusriemer/blockwerkzeug. 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` ensures that the most recent version of all images are available locally. If you don't pull the images first, the `run-dev` target might decide to build the required images locally instead.
- `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.

8.2.3 Frequent Issues and Error messages

These issues happen on a semi-regular scale.

I don't have any programming languages or projects available You probably forgot to load the initial data. Run `make load-live-data` in the `server` folder.

I changed things in the database, but they don't show up in the browser Rails does some fairly aggressive query caching which can **really** get in the way. Sadly the easiest option to fix this seems to be a restart of the server.

I don't want to log in for every operation You can give `admin` rights to the `guest` user which enables you to do almost anything without logging in. To do so you may run the following command from the `server` directory:

```
make dev-make-guest-admin
```

I need a dedicated admin account, the `guest` user is not enough.

1) If you don't have a regular account yet: Register one. During development you may use the "developer" identity which does not even require a password.
2) Find out your User ID, this can normally be accessed via the user settings page.
3) Run the following command from the `server` directory:

```
bin/rails "blattwerkzeug:make_admin[<Your User ID here>]"
```

Alternatively (if your display name is unique): Open a Rails console and run the following command:

```
User.find_by(display_name: "<Your Display Name>").add_role(:admin)
```

In both cases you need to log out and log in again to refresh your current token.

The server wont start and shows `Startup Error: No cli program at "../client/dist/cli/bundle.cli.js` The server requires the `cli` version of the IDE to run. Create it using `make compile-cli` in the `client` folder. The server will make more then one attempt to find the file, so if the program is currently beeing compiled startup should work once the compilation is finished.

CHAPTER 9

Programming Guidelines

Please read through these guidelines when developing for BlattWerkzeug.

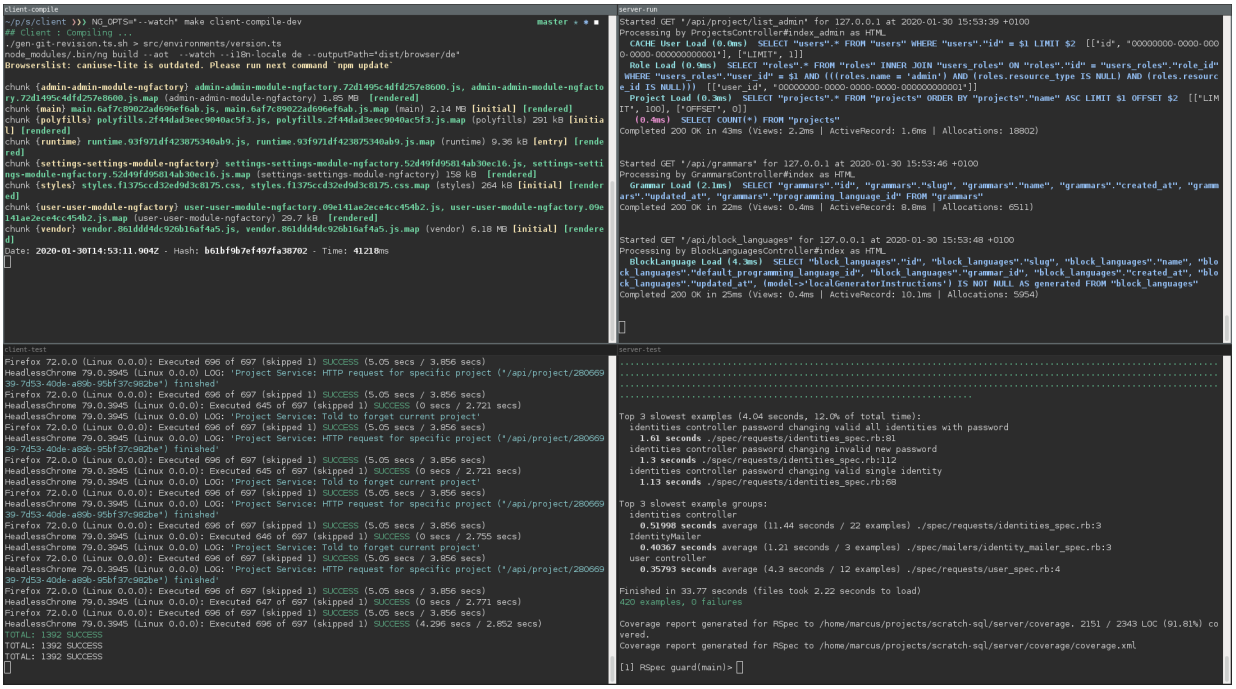
9.1 Code formatting

Code-formatting for the `client` folder is automated and enforced by prettier. You may either run `make format-code` in the `client` folder manually or setup your IDE or editor of choice to run `prettier` automatically: Official guidelines for editor integration.

9.2 Typical development setup

It is recommended to use four terminal windows when developing, that each show the output from one of the following `Makefile` targets:

- Targets in the `client` folder:
  - Run `make client-compile-dev` in the `client` folder. This will 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.
- Targets in the `server` folder:
  - Run `make reset-live-data dev-make-guest-admin run-dev` if you have pulled any seed data changes that need to be reflected. The `dev-make-guest-admin` target is optional, but very convenient during development.
  - Run `make test-watch` to continously run the server testcases in the background. This requires a running PostgreSQL database server.



By using this setup you ensure that all tests are run continously, no matter whether you are working on the server or on the client.

9.3 Tests, the CI-server and code coverage

Every `git push` will trigger a build on a CI-server that uses the Docker-images that are part of the repository. But you can (and should!) run the testcases locally, preferably all the time, but at least before a commit.

9.3.1 Client side testing

Calling `make test` in the `client` folder will run the tests once against a headless instance of Google Chrome and Firefox. Technically this works by passing the `--headless` flag when starting the browser, which suppresses any windows that would normally be shown.

- `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

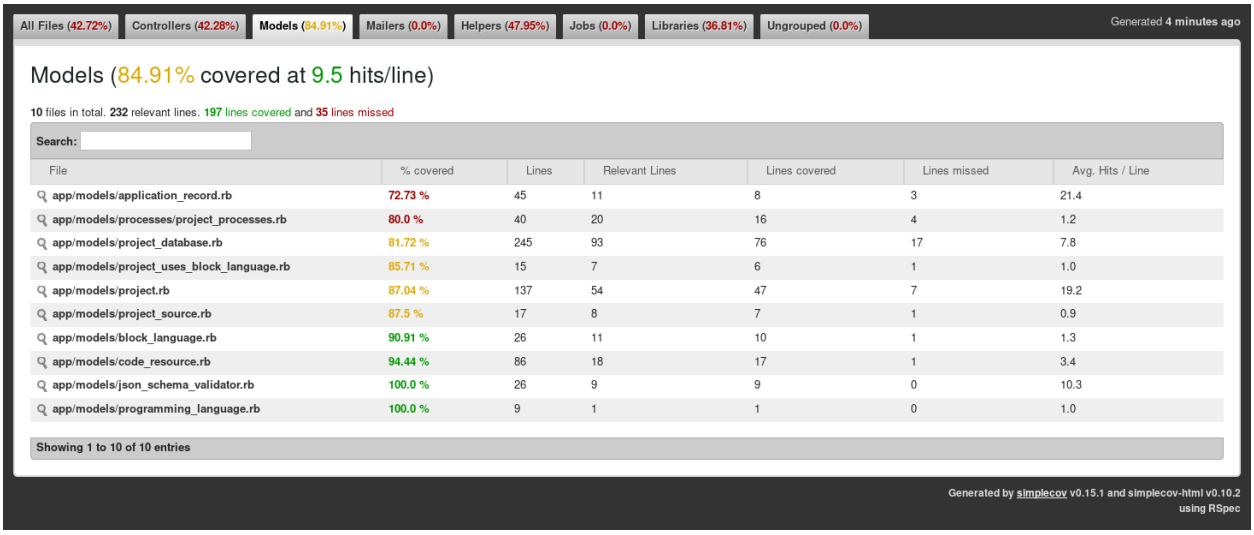
83.56% Statements 1540/1843 **75.13%** Branches 438/583 **75%** Functions 387/516 **83.96%** Lines 1419/1690

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)

9.3.2 Server side testing

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:



9.4 Description files and JSON schema

The server and the client need to agree on "over the wire" `JSON`-objects in order to function properly. As the server is using Ruby on Rails and the client is written in Typescript, this gap is bridged using JSON-schema. All of the relevant schemas live in the `schema/json` folder.

9.4.1 Updating schemas

The schemas are generated using an automatic conversion process based on Typescript `interface` definitions. By convention every file that ends on `description.ts` is a file that is meant to be used as an input for schema generation. If such a file is edited, the affected schemas need to be regenerated by running `make all` in the `schema/json` folder.

9.4.2 Rules for `description.ts` files

- They define interfaces that are used when communicating between server and client.
- They must not import complex libraries such as `rx.js`, `Angular`, ...
- Because other parts of the projects may import such complex libraries, the `description.ts` files may only import other `description.ts` files.

9.4.3 Using JSON schema to validate client requests

When clients want to make a request to the server, they usually need to construct a `object` that satisfied the relevant description-`interface`. For this part of the request, Typescript ensures just fine that the data that is send over will actually be understood and valid.

On the server a `controller` must use `JsonSchemaHelper#ensure_request`. As with any helper, the `JsonSchemaHelper` must be referenced via `include`. If a specific controller requires a request to be made with a `TableDescription` document, the source code would read as follows:

```
table_description = ensure_request("TableDescription", request.body.read)
```

The `ensure_request` method will take any string, parse it and validate it against the required schema and then return the corresponding hash with the data. If the given string is syntactically valid `JSON` but does not conform to the schema, an exception is raised and the request is aborted with a `401 Bad Request` response.

This procedure ensures that at least the **structure** of the passed in document is sound. There should be no need to re-validate the structure on the server or to program defensively in order to mitigate missing keys. Even if somebody sends request with arbitrary garbage, these should be filtered out by `ensure_request`.

9.4.4 Using JSON schema to validate server responses

Because the client is only expected to work with a conforming server, there is no response-validation infrastructure in place during the "nomal" execution of the client. Instead the results of `HTTP`-requests are simply casted to the expected `interface`.

The server side `request` tests ensure that the server responds with conforming documents. A custom rspec validator `validate_against` should be used with every testcase that expects a specific response.

9.4.5 Using JSON schema to validate models

The backing PostgreSQL database uses a few tables that store `jsonb`-blobs. After all, PostgreSQL is the best NoSQL database that is available The `jsonb`-columns are used for complex and self contained data structures such as the syntaxtrees for a code resource. To ensure that the database does not degenerate overall, the custom `json_schema` validator for Active Model ensures the validity of all stored blobs.

9.5 Loading and storing seed data

BlattWerkzeug comes with a complex set of required objects to work properly. This includes grammars, block languages, example projects, ... The "normal" Rails way of providing those objects via `db/seeds.rb` does not work for these structures at all: They are simply to complex to be meaningfully edited by hand.

The `Makefile` therefore exposes the `store-live-data` target which stores the current state of the programing languages and projects in the `seed` folder. This allows programmers to edit grammars, block languages and projects using the web-IDE and to persist those changes in the git repository.

Important: The YAML-files in the `seed`-folder are **very** prone to merge conflicts. Please make sure to only ever commit as small changes as possible. It is good practive to routinely use `make reset-live-data run-dev` when starting the server to ensure that your database-state is always up do date. If you run `store-live-data` from an old database state you may override newer changes that are part of the repository already.

9.6 Interactive Debugging

The preferred way to figure out the reason for undesired behaviour is by writing testcases: This ensures that the problem does not resurface later as a regression. But if you don't understand at all why something is going wrong, an interactive debugger is of course helpful.

9.6.1 Client Application

At least the normal development tools of Firefox and Chrome are capable of debugging the Angular application. Depending on your workflow, the `debugger` statement (Documentation at MDN) may be helpful to set breakpoints directyl from your editor of choice.

9.6.2 Client Tests

You may run the testcases interactively by surfing to http://localhost:9876/debug.html while `make test-watch` is currently running. This will take you to a page that runs all activated testcases directly in a browser.

CHAPTER 10

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.

10.1 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`

10.1.1 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.

10.2 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.

10.3 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`

10.4 Example configuration files

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

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

```
development:
  name: "Blattwerkzeug (Dev)"
  data_dir: <%= ENV['DATA_DIR'] || '../data/dev' %>
  client_dir: ../client/dist/browser
  schema_dir: ../schema/json
  project_domains: ["projects.localdomain:9292"]
  editor_domain: "lvh.me:9292"
  mail:
    default_sender: "BlattWerkzeug <system@blattwerkzeug.de>"
    admin: "Marcus@GurXite.de"
  ide_service:
    exec:
      node_binary: /usr/bin/node
      program: <%= ENV['CLI_PROGRAM'] || '../client/dist/cli/bundle.cli.js' %>
      mode: one_shot
  seed:
    data_dir: ../seed
    output: true
  sentry:
    dsn: <%= ENV["SENTRY_DSN"] %>
  auth_tokens:
    access_token: 180 # 3 minutes
    refresh_token: 432000 # 5 days
    access_cookie: # Is empty because of the session duration
    refresh_cookie: 1209600 # 14 days
  auth_provider: ["Identity::Google", "Identity::Github", "Identity::Password",
↪"Identity::Developer"]
  auth_provider_keys:
    google_id: <%= ENV['GOOGLE_CLIENT_ID'] %>
    google_secret: <%= ENV['GOOGLE_CLIENT_SECRET'] %>
    github_id: <%= ENV['GITHUB_CLIENT_ID'] %>
    github_secret: <%= ENV['GITHUB_CLIENT_SECRET'] %>
  seed_users:
    guest: "00000000-0000-0000-0000-000000000001"
    system: "00000000-0000-0000-0000-000000000002"
```

(continues on next page)

(continued from previous page)

```
test:
  name: "Blattwerkzeug (Test)"
  data_dir: "../data/test"
  client_dir: ../client/dist/browser
  schema_dir: ../schema/json
  project_domains: ["projects.localdomain:9292"]
  editor_domain: "localhost.localdomain:9292"
  mail:
    default_sender: "BlattWerkzeug <system@blattwerkzeug.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:
      node_binary: /usr/bin/node
      program: <%= ENV['CLI_PROGRAM'] || '../client/dist/cli/bundle.cli.js' %>
      mode: one_shot
  seed:
    data_dir: ../seed-test
    output: false
  auth_tokens:
    access_token: 180 # 3 minutes
    refresh_token: 432000 # 5 days
    access_cookie: # Is empty because of the session duration
    refresh_cookie: 1209600 # 14 days
  auth_provider: ["Identity::Google", "Identity::Github", "Identity::Password",
↪"Identity::Developer"]
  auth_provider_keys:
    google_id: 'GOOGLE_CLIENT_ID'
    google_secret: 'GOOGLE_CLIENT_SECRET'
    github_id: 'GITHUB_CLIENT_ID'
    github_secret: 'GITHUB_CLIENT_SECRET'
  seed_users:
    guest: "00000000-0000-0000-0000-000000000001"
    system: "00000000-0000-0000-0000-000000000002"

production:
  name: "Blattwerkzeug"
  data_dir: <%= ENV['DATA_DIR'] || '../data/prod' %>
  client_dir: ../client/dist-live/browser # Beware, this is *not* the normal output␣
↪folder
  schema_dir: ../schema/json
  project_domains: ["blattzeug.de"]
  editor_domain: "blattwerkzeug.de"
  mail:
    default_sender: "BlattWerkzeug <system@blattwerkzeug.de>"
    admin: "Marcus@GurXite.de"
  ide_service:
    exec:
      node_binary: /usr/bin/node
      program: <%= ENV['CLI_PROGRAM'] || '../client/dist-live/cli/bundle.cli.js' %>
      mode: one_shot
  seed:
```

(continues on next page)

(continued from previous page)

```
    data_dir: ../seed
    output: true
  sentry:
    dsn: <%= ENV["SENTRY_DSN"] %>
  auth_tokens:
    access_token: 180 # 3 minutes
    refresh_token: 432000 # 5 days
    access_cookie: # Is empty because of the session duration
    refresh_cookie: 1209600 # 14 days
  auth_provider: ["Identity::Google", "Identity::Github", "Identity::Password"]
  auth_provider_keys:
    google_id: <%= ENV['GOOGLE_CLIENT_ID'] %>
    google_secret: <%= ENV['GOOGLE_CLIENT_SECRET'] %>
    github_id: <%= ENV['GITHUB_CLIENT_ID'] %>
    github_secret: <%= ENV['GITHUB_CLIENT_SECRET'] %>
  seed_users:
    guest: "00000000-0000-0000-0000-000000000001"
    system: "00000000-0000-0000-0000-000000000002"
```

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

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

development:
  <<: *default
  database: esqulino_dev

test:
  <<: *default
  database: esqulino_test

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

10.4.3 Example `systemd` configuration

Listing 1: blattwerkzeug.service

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

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

[Service]
```

(continues on next page)

(continued from previous page)

```
# 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
```

10.4.4 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;

        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';
```

(continues on next page)

(continued from previous page)

```
    }

    # 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 11

Seed Manager

This part of the documentation is aimed to describe the store and load procedure of the seeds. It also describes the design part of it and how is it implemeted. Seed Manager can be extendted easily for a new model with felxible extention tecnique.

11.1 Store Procedure

Store procedure is a Service provided by the application which is used to call to store data for a particular Model (e,g. Project, Grammar..) with all the dependencies. It follows a simple pattern which evolves from `Seed::Base`. `Seed::Base` is the parent class where all the necessary methods are declared.

11.1.1 Naming Convention and necessary configuration

Seed Service follows a naming convention for the sake of the design which is part of Seed Module. `<model_name>_seed.rb` E,g. Seed class `project_seed.rb`, `project_uses_block_language_seed.rb`

Seed class takes three params `seed_id` which is mandatory a param must be passed as argument and `dependencis` is optional param only passed as argument if the Seed Model has dependencies and `defer_referential_checks` has default valu `false` unless any seed class privides other values based on the seed model's foreign_key_constraints.

`seed_id` could be `uid` of the Model or Model object.

The instance varible `seed_id` is the id of the Model that will be stored or processed.

- **other configuration parameters**
  - `SEED_IDENTIFIER = Project` is the name of the model
  - `SEED_DIRECTORY = "projects"` is the seed directory to store the seed
- optional dependencies

```
def initialize(seed_id)
  super(seed_id, dependencies = {
    ProjectUsesBlockLanguageSeed => "project_uses_block_languages",
    CodeResourceSeed => "code_resources",
    ProjectSourceSeed => "project_sources",
    ProjectDatabaseSeed => "project_databases",
    ProjectDatabaseSeed => "default_database",
  }, defer_referential_checks = true)
end
```

Seed are stored in a yaml file with a prefix of `seed_id` in corresponding directory

all the dependencies will be stored in its own `SEED_DIRECTORY` and it will create a dependency manifest *seed_id-deps.yaml* in the parent directory which contains a set of three idential value, `seed_path`, `seed_id` and `seed_name`. seed name is the seed model name.

Images and sqlite databases are stored in respective `SEED_DIRECTORY` with the corresponding seed_id

11.1.2 Call to store a seed

After seed class is defined according the above configuration and naming comvention (encouraged to follow), one can start stoing the data. e.g: `Seed::ProjectSeed.new(Project.first/Project.first.id).start_store`

Seed class can handle both Object or Object id

start_store calls `store` method which takes a Set object as argument. Which has been used for storing dependencies.

```
def store_dependencies(processed)
  dependencies.each do |dependent_seed_name, seed_model_attribute|
    data = seed.send(seed_model_attribute)
    to_serialize = (data || [])
    if not to_serialize.respond_to?(:each)
      to_serialize = [to_serialize]
    end
    to_serialize.each do |dep_seed|
      dependent_seed_name.new(dep_seed)
        .store(processed)
    end
  end
end
```

`seed` is the Model object we are storing either provided as constructor arguments or it calls a `find` on Model if provided seed_id is a id.

`dependencies` hash contains `{key => value}` where key is dependent seed and value is the model attribute to call the on the parent model to get all relative records.

if the return data is not an array incase it has only one record its need to be serialized. And then each record has passed to store with corresponding seed model.

`processed` is a set param with three values as the `store` method is designed to break the circular dependencies

`after_store_seed` hook is called after `store_seed` to enable seed classed to override this method if something like image or database needs to be stored after seed is stored.

```
def store(processed)
  if processed.include? [seed_directory, seed.id, self.class]
```

(continues on next page)

(continued from previous page)

```
  else
    store_seed
    after_store_seed
    processed << [seed_directory, seed.id, self.class]
    store_dependencies(processed)
  end
  if dependencies.present?
    File.open(project_dependent_file(processed.first[0], processed.first[1]), "w") do␣
↪|file|
      YAML::dump(processed, file)
    end
  end
end
```

Method itself describes the steps `processed.first` contains the parent class information

If the Seed does not have any dependencies, no problem as the default value of the `dependencies` is an empty array.

11.1.3 Store all seed of a seed class

To store all data, the example call will look like:

`Seed::ProjectSeed.store_all` or `Seed::GrammarSeed.store_all`

Its a class method which calls `store_all` method on Seed class, defined as:

```
def self.store_all
  self::SEED_IDENTIFIER.all.each { |s| new(s.id).start_store }
end
```

11.2 Load procedure

Load procedure of the seed also declared in `Seed::Base` class

It follows very simple pattern. It takes `seed_load_id` aka `seed_id` if seed_id is not a object itself.

and retuns files base name if any yaml file is provided to load

defined as:

```
def load_seed_id
  return File.basename(seed_id, ".*") if File.extname(seed_id.to_s).present? && File.
↪extname(seed_id.to_s) == ".yaml"
  return seed_id unless seed_id.is_a?(seed_name)
end
```

`load_id` is generated based on the type of `load_seed_id`, but always retruns an `id` regardelss of load_seed_id type

```
def load_id
  if load_seed_id
    if string_is_uuid? load_seed_id.to_s
      load_seed_id.to_s
    else
```

(continues on next page)

(continued from previous page)

```
      find_load_seed_id(load_seed_id.to_s)
    end
  end
end
```

As described in the Store Procedure, Seed class is configured with `SEED_DIRECTORY` and `SEED_IDENTIFIER`.

So When we start loading a particular seed we already know the seed directory

11.2.1 Upsert seed data

Upsert is meant to Insert or Update. As seed data is stored in a yaml file, we create a seed instance by loading the yaml file.

```
def seed_instance
  raise "Could not find project with slug or ID \"#{load_id}\"" unless File.exist?␣
↪seed_file_path
  YAML.load_file(seed_file_path)
end
```

Now upserting data from seed file path and after upserting it calls `after_load_seed` to load seed specific data

```
def upsert_seed_data
  raise RuntimeError.new "Mismatched types, instance: #{seed_instance.class.name},␣
↪instance_type: #{seed_name.name}" if seed_instance.class != seed_name
  Rails.logger.info " Upserting data for #{seed_name}"
  db_instance = seed_name.find_or_initialize_by(id: load_seed_id)
  db_instance.assign_attributes(seed_instance.attributes)
  db_instance.save! if db_instance.changed?
  db_instance

  Rails.logger.info "Done with #{seed_name}"
  after_load_seed
end
```

`seed_name` is the defined `SEED_IDENTIFIER` in the seed class

Code explains the steps of of intializng attributes for the model

It also handles dependencies by reading the the dependency manifest writtend during store procedure.

```
def load_dependencies
  deps = File.join seed_directory, "#{load_seed_id}-deps.yaml"
  deps = YAML.load_file(deps)
  deps.each do |_, seed_id, seed|
    seed.new(seed_id).upsert_seed_data
  end
end
```

Loads the `...-deps.yaml` file and takes each set data, where we need to take care of only last params one is seed_id and anoher is seed class.

Then it follwos the usual way to call `upsert_seed_data` method on seed instance.

Based on `defer_referential_checks` value it calls ‘‘ ActiveRecord::Base.connection.disable_referential_integrity‘‘ which takes the transaction block to enable deffered constraints.

Otherwise just runs the upsert and other methods. As a final step it moves the data from intermediate tmp storage to origianl storage defined in Project seed

To load a particular seed, the example call would look like:

`Seed::ProjectSeed.new(seed_id).start_load`

`start_load` is defined as follows

```
def start_load
  run_within_correct_transaction do
    upsert_seed_data
    dep = File.join seed_directory, "#{load_id}-deps.yaml"
    load_dependencies if File.exist? dep
  end

  if @defer_referential_checks
    db_instance = seed_name.find_or_initialize_by(id: load_id)
    db_instance.touch
    db_instance.save!
  end
  move_data_from_tmp_to_data_directory
end
```

It calls dependencies if only deps file are present in the seed directory

11.2.2 Load all seed data of a seed class

It's also a class method which calls `load_all` on seed class to be loaded, examle call will look like:

`Seed::ProjectSeed.load_all` or `Seed::GrammarSeed.load_all` and defined as:

```
def self.load_all
  Dir.glob(File.join load_directory, "*.yaml").each do |f|
    next if f =~ /deps/
    new(File.basename(f)).start_load
  end
end
```

Which excludes dependecy files because deps are extendted name of the the processed `seed_id` which is constructed based on availabilty of dependencies and `load_dependencies` method takes care of those files.

CHAPTER 12

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.

12.1 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.

12.2 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?

12.3 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 13

AST and Grammar Design Discussion

These are open (design) questions that should be answered by the thesis.

13.1 Structural and Visual Aspects

Currently the validation grammar and the visual grammar are stored in the same model. This is bad and must change, but whats a better approach?

Apart from beeing visually pleasing in its formal grammar representation it must also be meaningfully convertible in HTML. This sadly excludes the simplest possibility of simply inserting virtual linebreaks, because this doesn't work nicely with CSS flexboxes, see `minimal_indent.html`. for an example how this fails. A more or less straightforward HTML layout is proposed at `row-col.html`.

13.1.1 Example grammars (whithout visual aspects)

- Can't be meaningfully transformed into a block language, terminal symbols are missing

```
grammar "xml_1" {
  node "xml"."element" {
    prop "name" { string }
    children "elements" ::= element*
    children "attributes" ::= attribute*
  }

  node "xml"."attribute" {
    prop "name" { string }
    prop "value" { string }
  }
}
```

```
grammar "json_1" {
  typedef "json"."value" ::= string | number | boolean | object | array | null
  node "json"."string" {
    prop "value" { string }
  }
  node "json"."number" {
    prop "value" { integer }
  }
  node "json"."boolean" {
    prop "value" { boolean }
  }
  node "json"."object" {
    children allowed "values" ::= key-value*
  }
  node "json"."key-value" {
    children allowed "key" ::= string
    children allowed "value" ::= value
  }
  node "json"."array" {
    children allowed "values" ::= value*
  }
  node "json"."null" { }
}
```

13.1.2 Example XML grammar (with terminal symbols, current state)c

- Helpful: Syntax-aspects of XML are now known to the block editor
- Terminal symbols not enough information to turn into a block language: * Missing structural information line breaks or “horizontal” / “vertical” layouts. * No possibility to define “separators” between children
- Therefore: `children` command adds `seperator`

```
grammar "xml_2" {
  node "xml"."element" {
    terminal "tag-open-begin" "<"
    prop "name" { string }
    children "attributes" ::= attribute*
    terminal "tag-open-end" ">"
    children "elements" ::= element*
    terminal "tag-close" "<name/>"
  }

  node "xml"."attribute" {
    prop "name" { string }
    terminal "equals" "="
    terminal "quot-begin" "\"
    prop "value" { string }
    terminal "quot-end" "\""
  }
}
```

13.1.3 Idea: Seperate definition for "Visual Grammar"

- Is a visual grammar and must provide visualization for all instances of `node` mentioned in the visualized language.
- Adds a new type of command called `block`
- Allows to interpolate properties using `{{ }}`
- Inserts children using the `{{#children}}` directive.
- Problem: Nesting of `row` elements not straightforward.

```
grammar "xml_3" visualizes "xml_1" {
  block "xml"."attribute" {
    <row>{{name}}="{{value}}"</row>
  }

  block "xml"."element" {
    <row>&lt;{{name}}{{#children attributes, sep=" "}}&gt;</row>
    <indent>{{#children elements}}</indent>
    <row>&lt;{{name}}&gt;</row>
  }
}
```

13.1.4 Merging grammars

Sometimes languages are interweaved with one another, especially on the: `HTML` may contain `CSS` and `JavaScript`, many languages allow embedded `JSON` structures. It would possibly save lots of effort to allow grammars to be combined, e.g. to use the same `CSS` and `JavaScript` grammars that are already provided when describing `HTML`.

On a fundamental level, the grammars and the syntaxtrees have already been designed with this merging in mind: The `language` namespace is part of all definitions. The tricky part is the actual connection: How do we

13.2 Linked Trees

References between code fragments happen all the time: `HTML` documents reference `CSS` stylesheets, `JavaScript` files or other `HTML` documents, `Ruby` code `requires` other files, `C` loads them using `#include` ... The same should be possible with syntaxtrees in a hopefully generic manner, so that all block editors can either display the referenced resource inline or at least allow navigation to it.

13.3 Drop Target Resolution

- Each `block` introduces a new drop target, dropping something on it could mean "insert in here" or "append here".
  - Especially tricky with constructs like `if` were both operations are sensible.
- Current default:
  - Dropping on a block prioritizes the "append" operation, insertion happens only on demand * Great for `SELECT` of `SQL`: Children have different type then siblings
  - `children` introduce drop targets, may or may not be allowed to be empty

13.3.1 Implemented drop strategies

`allowExact` Allows a drop if the given drop location allows the insert, great (and default) for purposefully inserted drop markers.

`allowEmbrace` Allow the dropped thing to "embrace" the node at the given location, effectively replacing it. This is great for things like parentheses and unary or binary expressions, but can lead to bad conflicts with e.g. function calls (which are of course the general case of unary or binary expressions).

`allowReplacement` Allow the dropped thing to take the place of the node at the given deletion, effectively deleting it. This is useful if a location is a hole of length 1, e.g. replacing is the only syntactically sound option.

`allowAppend` Treat the drop as if it happened somewhere after the drop location (on a sibling level, not the child level). This is basically the default behavior of almost any visual and it is useful for lists of statements in imperative programming languages or lists of tables in SQL or lists of list items in JSON, ...

`allowAnyParent` Walks up the tree and checks all child groups of each parent whether an insertion would be possible. This is helpful in quite strongly typed grammars. In the current implementation of SQL it e.g. allows to drop the `SQL`-components (``SELECT`, `FROM`, `WHERE`, `GROUP BY`, ...)` virtually anywhere, because there is exactly one meaningful place that they could fit. In less strictly typed grammars this is probably not as useful.

13.3.2 Common drop problems and ambiguities

Appending vs Embracing in expressions in Lists If a non-leaf expression appears in a list of expressions, dropping something on that expression could mean `append` (add a new expression afterwards), `embrace` (e.g. negating the expression) or `insertAtChild` (e.g. adding a function call argument). The last option is not currently implemented as a strategy, because children are inserted using holes and `allowExact`. This strategy gets tricky however if e.g. a function (like `COUNT` in `SQL` or every function call in `JavaScript`) can take any number of arguments. The ambiguity regarding this can be reduced with a stronger type system.

Missing root nodes Synthetic nodes are a tricky thing to display, but are usually required at the root level. The "visual" roots of an `SQL` statement are either `SELECT`, `INSERT`, `UPDATE` or `DELETE`, but these components are actually child nodes of an `querySelect`, `queryInsert`, ... But the user doesn't want to drop those synthetic nodes ever, so there are two possibilities:

1. Create the synthetic root node together with the tree and never allow the user to change or delete it.
2. Don't actually drop a single node, but offer a list of semantically equivalent options.

Option #2 is what is currently implemented. When e.g. a `SELECT` component is dragged from the sidebar, two trees are actually tested for dropping: Nothing but the `SELECT` node or the `SELECT` node wrapped in the synthetic `querySelect` root node.

Type changes on dragging Dragging e.g. a function definition into a statement could be interpeted as "call this function". This however requires a change of the dragged type. The same happens in `SQL` when dragging a named expression from the `SELECT` component: The user probably doesn't want to insert `<expr> as <name>` into the `GROUP BY` component, but reference `<name>` there.

13.3.3 Dealing with ambiguity

Combing the drop strategies mentioned above may result in more then a single operation that could be carried out. It is probably not possible in all cases to resolve every ambiguity automatically, so this requires at least a nice UI.

- A simple but sort of "brutal" version would be to simply show a modal popup with all alternatives. The minimal implementation of this is very straightforward, as the validation process generates trees for all strategies anyway. Quite a lot nicer would be a "diff" of the trees and then only the subsequent display of differences to chose from.

- A possibly nicer version would be to leave "drop ghosts" in the tree: Instead of a single proper node, multiple feint 'ghost node' are inserted into the tree. These nodes require one more user interaction (e.g. a click) to actually be manifested into a proper node. This manifestations also removes all other ghosts that could possibly have been inserted, the user has therefor cleared up the ambiguity.