Sysrevving

Gjalt-Jorn Peters

2024-02-26T00:00:00+01:00

Table of contents

Preface

ℹ Note

Note: this is a living book; i.e. it is still very much a work in progress. The most recent version will always be available at https://sysrevving.com.

The SysRevving book is an open access book about doing open and systematic systematic reviews using open source tools. That is not a typo: it really says systematic systematic reviews, reflecting how this book aims to make systematic reviews as systematic as possible. This is tied in with the aim of making them as open, transparent, and machine-readable as possible.

The scientific literature is rapidly expanding, and keeping up-to-date by reading new journal issues as they come out has been impossible for many years already. As a consequence, two practices are becoming increasingly common: PhD. candidates starting their project with a systematic review to get a hopefully unbiased overview of the literature in a field, and conducting living systematic reviews that can be updated with relatively little effort to provide an up-to-date database to answer specific questions.

At the same time, the workflow adopted in many systematic reviews is often partly designed anew for each systematic review. Decisions, procedures, and documentation are often only human-readable and scattered over multiple files, sometimes even in proprietary file formats or formats used by free but not free/libre open source software (FLOSS). This book aims to support a systematic workflow geared towards optimizing transparency and machine-readability.

The idea is that optimizing transparency and machine-readability means that systematic reviews can ultimately become more scalable, interoperable, and extensible. This in turn means that there is less wasted effort and eventually, systematic reviews can be accelerated ('revved up', so to speak).

If you prefer, you can download the PDF version or the Ebook version (.epub) of this book.

You can cite this book as:

Peters, G.-J. (2023) SysRevving: A Guide to Open Systematic Reviews. https://doi.org/k9xc

1 Types of reviews

Scoping reviews, meta-analyses, systematic reviews, narrative reviews, conceptual reviews.

1.1 Narrative Reviews

1.2 Conceptual Reviews

1.3 Systematic Reviews

Systematic reviews differ from conceptual or narrative reviews in that they can afford the right to make strong statements by virtue of the employed procedures: those can and should be extremely rigorous, systematic, transparent, and reproducible.

This means that as much as possible, the decisions that are taken must be clearly documented and justified, so that the inevitable biases that the research team bring to the project can be taken into account when interpreting the results. It also means that the preparation phase is vital.

1.4 Meta-analyses

Meta-analysis is a homonym, referring simultaneously to a statistical approach and to a class of systematic reviews. The statistical approach comprises techniques to quantitatively synthesize multiple estimates of the same population parameter. Some of these techniques are relatively simple (such as when synthesizing two or more correlation coefficients), and some are very sophisticated (such as when using multi-level meta-regression).

The class of systematic reviews known as meta-analyses are those systematic reviews where the research question can be answered by quantitatively synthesizing a set of estimates, and where the heterogeneity of those estimates is sufficiently low to warrant such quantitative synthesis. Confusingly, the estimates of that heterogeneity require conducting a meta-analysis (the statistical approach). This means that in systematic reviews where the reviewers aim to quantitatively synthesize multiple estimates, but where those estimates turn out to exhibit so much heterogneiety so as to preclude a statistical meta-analysis, a statistical meta-analysis is conducted nonetheless to obtain those heterogeneity estimates. The result is a systematic review that is not a meta-analysis but that does include a statistical meta-analysis.

In such situations, the reviewers have to synthesize the estimates in another way, often resorting to qualitative integrations or to using visualisations. An insightful visualisation can be a forest plot, a visualisation typically used by statistical meta-analyses to illustrate how the synthesized estimate compares to the estimates from the separate studies – but then omitting the synthesized estimate.

1.5 Scoping Reviews

Scoping reviews or evidence maps (depending on who you ask, these can be the same or slightly different) differ from most types of systematic reviews in that they don't answer substantive research questions (note that when I use "systematic reviews", that also includes meta-analyses). Instead, they provide an overview of the scope of the literature: in a sense, they can tell you which research questions *can* be answered with systematic reviews.

Where most systematic reviews synthesize the evidence itself, often aiming to provide a more conclusive answer to the same or similar research questions as the included primary studies asked, scoping reviews synthesize the metadata about that evidence. They can tell you things like when most studies were conducted; which (sub)topics received most attention when; which study designed were used and whether that was associated to (sub)topic; how studies were distributed geographically; which sample sizes were common; whether any of those variables shows trends over time; et cetera.

Scoping reviews also produce an extensive database of literature, and are an excellent starting point for focused systematic reviews. Those also become much easier to plan, since you'll know how many studies are available. Depending on the comprehensiveness of your scoping review's extraction, you may even be able to skip the search and screening phases of those systematic reviews, since you already know which articles to include. Especially in combination with a decentralized approach to extraction, this means that scoping reviews can enable very efficient mapping of the literature.

Part I

Planning

2 Introduction to Planning

2.1 Overview

Planning a systematic review works in the opposite order of conducting it. Specifically, when planning, you and your research team have to achieve consensus on the following matters, in this order:

1) The goals and/or research question(s);
2) As a function of this, the entities you will extract (see the Planning: Extraction chapter, Chapter 5 in this version of the book);
3) As a function of these entities and the goal/research questions, the exclusion criteria you will use during screening;
4) As a function of the goal/research questions and the exclusion criteria, your search strategy, which includes:
    1) the conceptual form of the query you will use;
    2) which database(s) and interface(s) you will use;
    3) the conceptual query translated to each database/interface combination;
    4) additional strategies, e.g. forward and backward citation searches;

This will determine the scope of your review. Although these steps depend on the previous steps' output, in practice, this process is often nonlinear and iterative. For example, you often test draft queries in your interface/database combinations to see how many hits you obtain, potentially deciding to adjust your exclusion criteria or even your goals or research questions depending on what you find to ensure that the systematic review stays within the scope determined by your resources (time, funding).

2.2 Be explicit for redundancy, transparency, and future you

To err is human, and therefore, in scientific endeavors, it is best to never count on a single human not erring. The solution to this is two-fold. First, don't let humans perform tasks that computers can perform; and second, implement redundancy. That means that for tasks that have to be performed by humans, always have at least two people perform every task independently and check for consistency in the results.

If you can afford such redundancy, there is a clear penalty for sloppy planning. If your definitions, descriptions, and instructions leave room for interpretation, the laws of probability decree that that room will inevitably be taken sooner or later. This will manifest as heterogeneous results that will be labor-intensive to reconcile. In a non-trivial proportion of cases, such divergent results may prove almost impossible to reconcile, as they may bring to light fundamental problems with the tasks, definitions, and procedures you specified during your planning.

However, in many cases, full redundancy is not practically attainable. For example, when a systematic review is conducted in the context of a bachelor's thesis, a master's thesis, or a PhD. thesis, only one screener and one extractor may be available. Similarly, in many projects, having multiple independent synthesists is not feasible. In such cases, sloppy planning and sloppy documentation of the planning is not penalized as explicitly and as acutely.

Therefore, in such cases, it is particularly important to pay special attention to clearly documenting your plans, definitions, decisions, and their justifications. In addition, it is important to not forego developing instructions for each task as if you were not the only person who will conduct them. Although redundancy may not require this, there are two reasons to do this nonetheless.

First, as the transition towards open science has shown, there is much to gain from exercising transparency in science, both epistemologically and operationally. Epistemological benefits include error-detection, easy and accurate identification of risks of bias, and availability of process information for meta-scientific interrogation. Operational benefits include facilitating learning from other researchers and prompting more elaboration through the awareness that one works in public.

Second, systematic reviews tend to be both very valuable and take a lot of time, and the process tends to be very similar every time. These characteristics mean that you will likely come back to your earlier documentation and plans, either to remember what exactly you did and why, or because you want to adapt and re-use some elements of your process. You will save Future You a lot of time and effort by exercising some minimal hygiene throughout your project in terms of data management and clear documentation, including instructions that may seen unneccessary at the time.

3 Research Questions

The research questions guide much of the planning (and so, execution) of a systematic review. I say "much" because when you're planning a systematic review and it's not necessarily a one-shot endeavour, you will often want to anticipate as many future needs and wishes as possible, which extends your planning beyond your current research questions.

Still, any systematic review will be conducted with a specific initial goal in mind, and because that goal will often be obtaining one or more answers, the questions to be answered are a useful way to structure the planning.

These research questions will always contain one or more concepts. Each of these concepts will relate to one or more entities to extract (see below). Once you have decided on your research question, therefore, you can decide on the entities to extract.

However, in practice this process is nonlinear and iterative. Any given research question (or more accurately, any given set of entities to extract) implicitly determines the scope of the review, because the exclusion criteria are based on the research questions and the entities to extract, and the search strategy (e.g. the query) is based on the research questions, the entities to extract, and the exclusion criteria. As such, there is always some correspondence between the research question and the number of sources that your search strategy will yield (and that will have to be screened).

Therefore, you will usually develop all of these in parallel. For example, it is common to test different queries until the number of hits is feasible given the available resources. Depending on your screening capacity, you may be forced to revise and limit the scope of the research question(s). Similarly, if your query yields very few hits, you may want or need to broaden it (and therefore, broaden your research question(s)) to eventually obtain worthwhile results.

3.1 Systematic Reviews

3.2 Scoping Reviews

Research questions in scoping reviews ask what researchers did. They can concern anything from whether different geographical regions prioritize different topics, whether sample sizes increased or decreased over time, which definitions researchers use, and which measurement instruments researchers use, via things like which study designs are used to answer which types of questions and how paradigms change over time to whether shifts occurred in researchers' underlying philosophy of science or epistemological perspectives.

4 Planning the Synthesis

Where in primary research, the term used for the process by which one arrives at answers to a research question based on the collected data, in a systematic review, this process is called synthesis. This nicely captures the aim to combine information from multiple sources to yield a (hopefully coherent) overview.

Synthesis of systematic review results shares many characteristics with analyses in primary research. In both cases, the process consists of dozens to thousands of subjective decisions; in both cases, the process can be challenging and complicated; and in both cases, the way the process is challenging and complicated depends on the type of data being processed. For systematic reviews, syntheses tends to be easier as sources are more similar.

For example, a relatively simple synthesis could be a meta-analysis of studies that all have the same design and used the same measures for the same variables, for example when aggregating randomized controlled trials for a specific COVID-19 vaccination. In such scenarios the studies often share the same ontological and epistemological perspectives, and a single effect size estimate can often be extracted from each source, all in the same metric. In addition, variation in extracted effect sizes most likely reflects sampling variability and contextual factors. Both can be statistically modeled (assuming those contextual factors were extracted).

An example of a relatively complicated synthesis is a systematic review that provides an integrative overview of a topic over multiple study types, for example everything that is known about why people

5 Planning the Extraction

5.1 Introduction to extraction

The extraction is the stage where the "data" are extracted from the identified *sources* (see Chapter 19). This means that the information from or about the included sources has to be stored in an extraction script file.

In a systematic review, the extraction and synthesis stages are the hardest (unless a meta-analysis is possible, in which case the extraction stage is the hardest). This is because the information you want to extract will often be ambiguous, and sometimes it will not be available at all. This ubiquitous ambiguity means that the task of extracting information is typically not a matter of copying information over: instead it's more like playing detective.

5.2 Entities

Planning the extraction is also the hardest part of planning a systematic review. Planning the extraction means specifying the R extraction script, which requires specifying the entities to extract, how they are hierarchically organized, and which value templates each entity uses.

An entity is anything that has to be extracted from a source, such as the year something was published, its authors, a definition that was used in a source, a theory that was studied, a study design, a sample size, a measurement instrument, the literal text of measurement instrument items, expressions from interview participants, effect sizes, or things like the number or figures, tables or words in a source.

During the planning phase, you decide which entities you want to extract and how. A garden-variety entity represents one type of thing you want to extract from your included sources. You will define them in the Rxs specification, a spreadsheet (see below for details).

5.2.1 Identifiers and titles

For every entity, you will choose an identifier, a title, and a description. The identifier is a unique machine-readable name for your entity. This will allow you to easily select all extracted data for a given entity during the synthesis phase. Identifiers can only contain lower- and uppercase Latin letters, Arabic numerals, and underscores, and must always start with a letter. [1] Titles are the human-readable equivalent, so basically just the name of the entity, without any constraints as to which characters you're allowed to use, but short.

5.2.2 Descriptions and extraction instructions

Descriptions are longer, and should contain at least two, and preferably three elements. First, the description should describe and define the entity. Since what you include here will be all that extractors, other researchers and interested parties, and Future You will have to go on, it pays to make an effort to be as explicit as possible.

Second, the description should contain explicit extraction instructions for the extractors — even if you yourself are going to be the only extractor (see "Be explicit for redundancy, transparency, and future you", Section 2.2 in this version of the book).

Third, ideally, the description should explicitly list one or more edge cases. Edge cases are examples of something that a source might contain where it is not obvious how it should be extracted correctly. By listing these and explicitly describing why that example should be extracted the way you specify, you help extractors (including Future You) understand better how you delineate your entity definition.

5.2.3 Values to be extracted

To specify which types of values can be extracted for each entity, value templates can be specified (see the Value Templates section below). For each entity, the valid values, default value, and examples specified in the value template can be overridden in the entity specification.

5.2.4 Hierarchical structure and container entities

Because the number of entities extracted from the sources in a systematic review can become quite large, and are often clustered together, entities have a hierarchical data structure. This means they form a tree: each source is a root that can have leafs and/or branches attached. In data tree terminology, terms from two vocabularies are mixed: tree terms, such as root, branch, and leaf; and family terms, such as parent, child, descendants, and ancestors.

To familiarize yourself with these terms, consider the tree in Figure 5.1.

[1] If you're already familiar with regular expression, the regular expression is `[a-zA-Z][a-zA-Z0-9_]*`. If you're not already familiar with regular expressions: they're an extremely powerful tool to describe, search for, and replace text fragments, well worth at least a brief acquaintance.

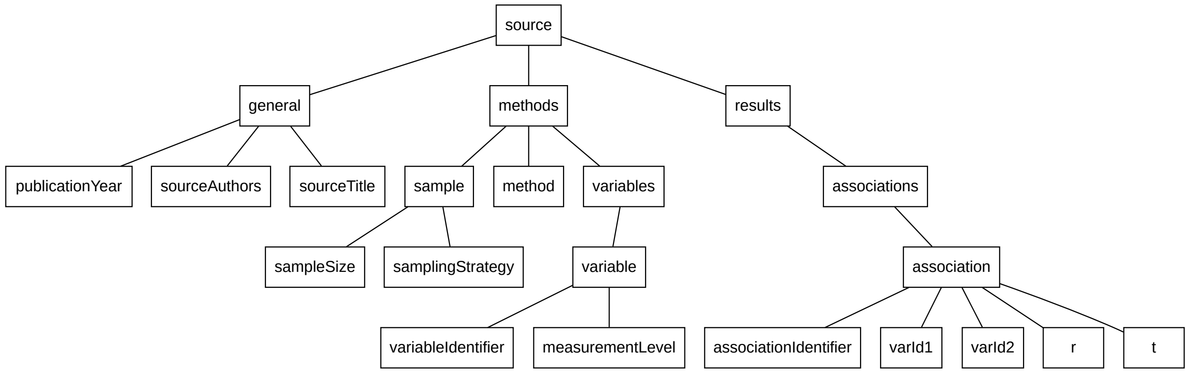


Figure 5.1: An example data tree with entities.

In this tree, the `source` itself is the root where all entities are attached. The three *container entities* attached to the root are `general`, `methods`, and `results`. These container entities are used to organise other entities: nothing is extracted for those contained entities themselves, they just function to organise and represent their children. The children of the `general` container entity (`publicationYear`, `authors`, and `title`) are themselves leaves: they have no further descendants.

Of the entities specified as children of the `methods` container entity, the `method` entity doesn't have descendants either: that entity is also a leaf. The `sample` entity does have two children (i.e. is a branch), `sampleSize` and `samplingStrategy`, each leaves themselves (i.e. without children). The `variables` entity has one child that is itself a branch (`variable`), which has two children: `variableIdentifier` and `measurementLevel` (both leafs).

Finally, the `results` container entity contains one container entity called `associations`, which contains another container entity (i.e. a branch) called `association`, which contains three regular entities (i.e. leafs): `varId`, `r`, and `t`.

The position of an entity or container entity in the Rxs tree is specified by its parent, where the entity identifier of the parent container entity is specified.

5.2.5 Repeating entities

Sometimes, an entity, entity container, or clustering entity (see the next section) can potentially be extracted multiple times. For example, a source may report on multiple samples or may report multiple effect sizes. Therefore, some entities are repeating entities, which means the extraction script will be set up such that the corresponding lines can be copy-pasted multiple times. Normally, only clustering entities will be repeating. This will be explained in more detail in the next section.

5.2.6 Clustering entities ('lists')

In addition to container entities, that themselves contain no extracted data but are used to organize other entities, there are *clustering entities*. You can consider clustering entities as a special type of container entity that only contains leaf entities that are closely related to each other. In the extraction script template, these *clustered entities* (i.e. the leaf entities in a clustering entity) are placed on successive lines, with their titles, descriptions, value template descriptions, and examples all concatenated in one line after the bit where the entity itself is extracted.

There are two benefits to using clustering entities. First, they are more efficient during extraction, especially if the clustering entity is a repeating entity (see below). Second, `metabefor` has functions to supplement a clustering entity with entities from elsewhere in the extraction tree.

To illustrate this, again look at Figure @ref(fig:planning0extraction-treeIllustration). In this Rxs tree, there are two repeating clustering entities: `variable` and `association`. The `variable` clustering entity contains two clustered entities: `variableIdentifier` (a unique identifier for each extracted variable) and `measurementLevel` (the measurement level of this variable). The `association` clustering entity contains five clustered entities: `associationIdentifier` (a unique identifier for each extracted association), `varId1` (the identifier of the first variable, referring to a `variable` clustering entity by its `variableIdentifier`), `varId2` (the identifier of the second variable, also referring to a `variable` clustering entity by its `variableIdentifier`), `r` (a Pearson correlation coefficient), and `t` (a Student t value).

Both the `variable` and `association` clustering entities are repeating entities. This means that they can each be extracted multiple times by copy-pasting the relevant lines in the extraction script. Because each clustering entity has a unique identifier, they can be referred to, and each `association` clustering entity refers to two `variable` clustering entities.

Now, imagine a systematic review on gym membership, exercise, diet, and BMI. The extractor might encounter a source where they extract four effect sizes in four `association` clustering entities:

- the Pearson correlation between height and weight;
- the Pearson correlation between weight and daily energy ingestion;
- the Pearson correlation between weight and daily exercise; and
- the Student t value for the association between gym membership and daily exercise).

The extractor also specifies the measurement level for each variable in five `variable` clustering entities.

During synthesis, `metabefor` allows supplementing the `association` clustering entities with the information specified in the `variable` clustering entities using the unique identifiers specified in `varId` and `varId2` and then looking for the corresponding clustering entities with that identifier in their `variableIdentifier` entity.

This is a trivially simple example, but this functionality is very powerful to extract efficiently and with high fidelity, while retaining flexibility and easily recombining the extracted entities during the synthesis stage to ultimately obtain data frames that lend themselves well to the intended synthesis.

5.3 Value Templates

Value templates are an efficient method to define a type of data to be extracted. The example `metabefor` Rxs specifications contain a number of common value templates:

- `numeric`: Any valid number
- `numeric.multi`: A vector of valid numbers
- `integer`: Any valid whole number
- `integer.multi`: A vector of integers (i.e. one or more whole numbers)
- `integer.length4.multi`: A numeric vector of years
- `string`: A single character value
- `string.multi`: A character vector (i.e. one or more strings)
- `countrycode`: A character vector of the ISO 3166-1 alpha-2 country code(s)
- `categorical`: A string that has to exactly match one of the values specified in the "values" column of the Coding sheet
- `generalPresence`: Whether the thing being coded was present or not.
- `string.mandatory`: A single character value that cannot be omitted
- `string.entityRef.mandatory`: A string that specifies another entity and which MUST be provided
- `string.entityRef.optional`: A string that specifies another entity (can be missing, i.e. NA)
- `string.fieldRef.optional`: A string that specifies another field in another entity (can be missing, i.e. NA).
- `matrix.crosstab`: A table with frequencies; variable 1 in columns, variable 2 in rows; always work from absence/negative/less (left, top) to presence/positive/more (right, bottom)
- `string.identifier`: A single character value that is used as an identifier and so is always mandatory and can only contain a-z, A-Z, 0-9, and underscores, and must start with a letter.

Each value template specifies a unique identifier, a description, optionally the valid values that can be extracted, a default value to insert into the extraction script template, one or more examples, an R expression to validate the extracted value (which implements the descriptions in the list above), and an error to show if that validation fails.

5.3.1 Validation of extracted values

The R expression that is specified as validation has to be a logical expression that evaluates to a single `TRUE` or `FALSE` value. In this logical expression, you can use the placeholder `VALUE`, which will be replaced by the value that the extractor extracted, and the placeholder `<<validValues>>`, which will be replaced by whatever you specified in the `validValues` column of the vlaue templates worksheet. Note that if you want to specify multiple values in the `validValues` column, you have to separate them with character sequence `||` (so, space, pipe, pipe, space).

You can compare the `VALUE` the user supplied to other values and test its contents using a logical expression where you can use the following operators:

- `||`: a version of the "or" operator that takes two single values
- `&&`: a version of the "and" operator that takes two single values
- `|`: a version of the "or" operator that takes two (equal-length) vectors
- `&`: a version of the "and" operator that takes two (equal-length) vectors
- `!`: the "not" operator, placed in front of the expression ot negate

In addition, it is highly recommended to use parentheses (`(` and `)`) to explicitly specify the order in which you want the "sub-expressions" that together form the logical expression to be evaluated.

Finally, you can use R functions. Common functions you may want to use are the following:

- `is.na()`: tests for every element in the value, vector, or list you pass, whether it's missing ("`NA`") or not
- `is.numeric()`: tests for every element in the value, vector, or list you pass, whether it's a numeric value or vector
- `is.character()`: tests for every element in the value, vector, or list you pass, whether it's a character value or vector
- `length()`: returns the length of the vector (or `1` if it's a single value)
- `all()`: returns whether all values you passed are `TRUE`
- `any()`: returns whether at least one of the values you passed is `TRUE`
- `nchar()`: returns the number of characters in (each value of) VALUE

5.4 Details of the Rxs specification

The entities are specified in a spreadsheet called an Rxs specification. Rxs stands for R Extraction Script, and they are the machine-readable files that data from sources are extracted into.

They are in fact R Markdown files that can be rendered as-is, but that can also be imported using `metabefor`. These files are created by `metabefor` based on the Rxs specification.

An very minimal example of such as spreadsheet is available at https://docs.google.com/spreadsheets/d/1Ty38BS GJ6zzr7E3rC_vQNOMKe-uCvIuHs3c. A more extensive example is available at https://docs.google.com/spreadsheets/d/13MUf8qL4Zmc5V6AOvjO1GWeFCl4IaaSl2zUT-Kk9tQc. See the @ref(example-projects) chapter in the Appendix for a list of example projects.

A spreadsheet holding an Rxs specification has at least the following worksheets:

- `entities`: The specifications of the entities to be extracted in the systematic review.
- `valueTemplates`: The value template specifications: an efficient way to specify 'data types' for entities.
- `definitions`: Definitions of concepts used in the systematic review.
- `instructions`: Instructions for the extractors.
- `texts`: Texts to override `metabefor`'s default texts.

These will now briefly be described.

5.4.1 The `entities` worksheet

The `entities` worksheet has the following columns:

- `title`: A short human-readable label for the entity (basically its name).
- `description`: A longer human-readable description of the entity. Together with the value template descriptioin, this will form the instruction for the extractors, so make sure to clearly describe what they should look for in the sources.
- `identifier`: A machine-readable identifier for this entity. This may only contain lower and upper case Latin letters (`a-z` and `A-Z`), underscores (`_`), and Arabic digits (`0-9`), and must start with a letter. This is used to refer to extracted entities in the results, or when cross-referencing entities (e.g. in the `parents` column).
- `valueTemplate`: The identifier of the `valueTemplate` to use (see the `valueTemplates` worksheet).
- `validValues`: Overrides the `validValues` specified in the specified 'valueTemplate.
- `default`: Overrides the `default` value specified in the specified 'valueTemplate.
- `examples`: Overrides the `examples` specified in the specified `valueTemplate`.
- `parent`: The entity's parent entity: in the hierarchical tree of extracted entities, the parent is the entity that this entity will fall under. For example, in the Rxs specification for the the example tree shown above, the entities `samplingStrategy` and `sampleSize` each list `sample` in the `parent` column.

- `list`: If `list` is set to `TRUE`, that designates this entity as a *clustering entity*. That means that the entities it contains are *clustered entities* that are presented in the extraction script in a `list()`. This allows for more efficient extraction of the child entities. However, is also means that in the tree of extracted entities, these child entities (i.e. the clustered entities) cannot themselves have child entities. In other words, those child entities are all leafs on the tree.
- `repeating`: Set `repeating` to `TRUE` for entities that can be extracted multiple times. This is useful for, for example, effect sizes or other statistics, which can be extracted multiple times for a given source, but always have the same specifications.

These columns are also included, but contain functionality that is both quite advanced / abstract and not yet fully implemented in `metabefor`:

- `collapsing`: To be added.
- `recurring`: To be added.
- `recursing`: Set `recurring` to `TRUE` for entities that can recurse: that can contain themselves.
- `identifying`: Set to `TRUE` if this is entity if an identifier.
- `entityRef`: It is often useful to specify that extracted information relates to a specific entity (usually an entity container). In such cases, this column can be used to specify which entity is referred to. This is then used during validation to verify whether in the tree object, the value specified for this entity occurs as one of the values specified for the `entityRef` entities. For example, when conducting a meta-analysis, it is typically useful to extract the variables measured in a study as well as estimates for associations between those variables. Using `entityRef` entities, it is possible to extract the measurement instrument used for the relevant variables only once, and then refer back to those entities using the `entityRef` entity.
- `fieldRef`: [this is advanced functionality that still has to be implemented in `metabefor`] Sometimes, extracted information does not relate to another entity, but to one specific value for an entity specified in the `entityRef`. The `fieldRef` field allows you to specify the identifier of the entity within the entity referenced in the `entityRef` entity to which the parent entity pertains.
- `owner`: This entity's owner - specifying an owner signifies that all entities with that identifier must contain at least one entity with the current identifier.

5.4.2 The `valueTemplates` worksheet

- `identifier`: The unique identifier of this value template, used in the `entities` worksheet to specify that this value template should be applied to an entity. This must be a machine-readable identifier, and so may only contain lower and upper case Latin letters (`a-z` and `A-Z`), underscores (`_`), and Arabic digits (`0-9`), and must start with a letter.

- `description`: A description of this type of value. This will be shown in the Rxs template for every entity that this value template will be applied to. Specifically, extractors will see this description printed below those entities.
- `validValues`: Optionally, a list of valid values. Each value must be separated by double pipes (`||`). For example: `"Unknown" || "Present" || "Absent"` means that one or more of those three strings must be extracted.
- `default`: The default value inserted in the Rxs template.
- `examples`: Examples of extracted values.
- `validation`: An R expression to validate the extracted entity.
- `error`: An error message to show if the validation fails.

5.4.3 The `definitions` worksheet

Here, you can specify definitions that are important in your project. They will be inlcuded in the extractor instructions, together with the contents of the `instructions` worksheet. There are two columns:

- `term`: A term.
- `definitions`: The corresponding definition.

5.4.4 The `instructions` worksheet

Here, you can specify instructions for your coders.There are two columns:

- `heading`: A heading, which will be included as a heading in the instructions.
- `definitions`: The instructions that should be displayed below that heading.

5.4.5 The `texts` worksheet

This functionality has not been implemented yet, but it will allow overriding the default texts produced by `metabefor`.

- `textIdentifier`: A unique identifier for the text fragment.
- `content`: The text fragment that should be used.

5.5 Post-hoc entity specification: Txs specifications

Tabulated Extraction Sheet specifications

6 The Categorization-Coding Continuum

6.1 Categorization

For some entities, the potential values an extracted entity can take are knowable in advance in a given systematic review context. For example, when reviewing primary studies in humans or other animals, the sample size must be a positive integer (e.g. 1, 2, 3, …); and publication year will usually have to be a positive integer, often of four digits. In other cases, it is clear that free text will be extracted, for example, when author names or source titles are extracted.

For many entities, however, it is less obvious how to operationalize them. When something is extracted as free text, often as many unique values will be extracted as there are sources. This means that synthesis (i.e., "analysis", see below) first requires transformation of those values. A list of raw free text values cannot be synthesized: the strings of characters have no encoded meaning, and cannot be collapsed or summarized. Nothing can be calculated from a list of free text values; and if that list is used in a table, that table will have as many rows or columns as the list of free text values has different values. Especially in scoping reviews, where including hundreds of sources can be quite common, this often isn't feasible.

In many cases, this problem can be avoided by having extracters categorize that information during extraction. For example, imagine a scoping review into qualitative research practices in a given field. One of the entities that will be extracted is how the researchers coded the data. In this case, an infinite number of coding approaches can be used. Many textbooks on qualitative research use some categorizations to organize these. For example, coding can be categorized as "inductive" versus "deductive" or as "open" versus "axial". Like any categorization, these simplify reality, making it easier to deal with for humans. If this simplification is not problematic given the scoping reviewers' research question(s), they can choose to adopt one of those categorizations.

In that case, they would decide which categories to use (e.g. `inductive coding` and `deductive coding`, or two symbols representing these two categories, such as `1` and `2`) and specify clear coding instructions for each (often with special attention to edge cases). After extraction, instead of having one or several sentences of free text extracted for each source (where the original authors describe their coding approach), they would then have a list with only possible two values (e.g. `inductive coding` and `deductive coding`, or `1` and `2`). This lends itself to easy synthesis: the percentage of sources using inductive coding could easily be obtained, and it would be possible to answer questions such as whether that percentage seems stable over time, or differs between subdomains, or by geographical area.

6.2 Ambiguity

However, the extractors would also encounter sources where the authors used both types of coding - and they would encounter sources where a coding approach would be used that could arguably belong in either (or neither) category. There are two strategies to try and prevent such problems.

The first is developing very, very comprehensive coding instructions. If the scoping reviewers have a clear idea of all potential coding approaches, discussing all edge cases extensively in the coding instructions can ensure unequivocal (and correct) categorizations of most potential descriptions extractors can encounter in the sources. For example, the coding instructions can instruct extractors to categorize all sources using both inductive and deductive coding as "inductive" (or "deductive", depending on what makes sense given the scoping review's goals).

The second is putting a lot of thought into the categories that are used for each entity. For example, instead of using two categories, the scoping reviewers could add a third category `inductive and deductive coding`. They could also split the entity into two dichotomous entities, having extractors extract whether inductive coding was used into one, and whether deductive coding was used into another. By adding a third category `unclear` to each entity, ambiguous cases could be easily spotted - however, at the cost of no longer knowing what the extractor would guess if forced. That could be solved by adding more categories, for example extracting the entity `inductive coding` into categories `no`, `unlikely`, `likely`, and `yes`; or, alternatively, by adding a second entity that holds the extractor's confidence in the categorization.

6.3 The cost of categorization

Each of these solutions to the problems caused by reality (including researchers' decisions as extracted in scoping reviews) usually not being neatly organized into categories entail some costs. The more entities that are used to store the information extracted from the sources, and the more categories that are used for each entity, the less information is lost during extraction – but the more time and effort the extraction costs.

In addition, any categorization by definition means that what can be learned from the systematic review is limited to the "potential answer space" formed by what the systematic reviewers knew a priori. If a research question is "which coding approaches are used", and the entities that systematic reviewers extract into are `inductive coding` and `deductive coding` (both with categories `no`, `unlikely`, `likely`, and `yes`), then the synthesis can never result in conclusions about the proportion of sources where the researchers reported they used guinnea pigs, neural networks, or magic crystals for coding, even if a sizeable proportion of the sources reports those approaches. Each of these three types of coding approaches will instead be categorized as either inductive coding or deductive coding (or potentially both) – if the coding instructions are of sufficient quality, they will be categorized unequivocally and consistently, but still, a lot of information will be lost.

This can be problematic depending on the research questions. Often, what the systematic reviewers *do not* see coming a priori can be the most interesting. When the nature or scope of the "potential answer space" is not the thing of interest (i.e., the researchers are interested in where the set of included sources falls in that space), the costs of categorization can be zero or low. However, when it is not clear in advance how that space looks, researchers may not be able to afford categorization at extraction time. In that case, coding can happen after the extraction stage.

6.4 Coding after extraction

When coding after extraction, during extraction the only decision the extractors face is which fragments to extract. They don't need to interpret anything beyond identifying which part(s) of the source contain(s) the relevant information, which decreases the probability of errors. That interpretation then comes after.

The extracted original raw text fragments can then be exported to `.rock` files that can be coded using the Reproducible Open Coding Kit (ROCK) standard. The coded files can then be imported again and merged into the object holding all extracted data.

There are a lot of advantages to this approach. First, it makes the review much more transparent. It's easy for others to see which fragments were selected, and so what the results were ultimately based on. Second, it lends itself well to rigorous quality control: having a file with extracted fragments coded by multiple coders is relatively straightforward and 'cheap' (timewise), since the selection of the relevant fragments is often a large part of the task. Third, it scales very well: the tasks of selecting the relevant fragments and the interpretation of those fragments can be distributed between multiple extractors and coders. Fourth, closely related, it enables a decentralized approach, where different groups can work on different parts of a project. This means that it enables involving students or citizen scientists. Fifth, it provides flexibility regarding effort distribution over time. If twenty entities are extracted as free raw text fragments, reviewers can decide to start with coding the first five, which might be enough to answer their main research questions. The other fragments can then be coded later on, without delaying the rest of the project. Sixth, it allows relatively efficient re-coding using different categories, which is for example very useful when conducting living reviews, where insights about how to categorize can change over time.

There are also disadvantages to this approach. First, it costs more time to record raw text fragments (which requires copy-pasting, usually from PDFs which also means it often also requires some cleaning of the pasted text) than it takes to record a selection from a predefined set of categories. Second, experienced extractors develop competences that make them more efficient and more consistent over time. By cutting up the tasks and potentially distributing those over more people, this training effect decreases.

6.5 The categorization-coding continuum

Whether a given entity is extracted as a raw free text fragment or categorized during extraction, and if the latter, which categories are used and whether the entity is split up into multiple entities has to be decided in the planning stage. Changing this decision once the extraction has started is *extremely* expensive in terms of time, energy, and "error-prone-ness", which means that it is worthwhile to put a lot of thought into this decision for every entity.

In fact, together with which entities are extracted, how to extract each entity is the most important decision taken when planning a systematic review. These decisions determine for a large part how much time and energy the review will take, as well as how flexible the compiled database will be, how extensible the review will be, and whether the process is scalable and lends itself to decentralization.

Whether an entity should be extracted as raw free text fragments that are then later coded, or categorized during extraction, or any of the options in between (e.g. categorization into one entity with a second entity to specify raw text fragments in case of a misfit with the prespecified categories; or coding into predefined categories but using multiple entities and many categories to lose as little information as possible), depends on a number of things. For example, when few resources are available (e.g. time, people), extracting raw text fragments and coding afterwards may not be feasible. If conversely, if the reviewers aren't confident they can specify a well-defined set of mutually exclusive categories with clear coding instructions, they will have to extract raw text fragments and defer the category definition to the coding stage.

In addition, in one-shot reviews, some of the benefits of extracting raw text fragments and separating the categorization from the extraction dissappear, and the remaining benefits may not outweigh the costs. Conversely, when conducting a living review, being able to code extracted text fragments again at some point in the future using a different perspective, or having different coders code the text fragments with different goals and instructions can be useful.

In any case, given the importance of this decision, it is worthwhile to carefully document for each entity what the justifications are for its chosen position on the categorization-coding continuum. Later in the process, it is likely you will forget those, and you may even regret your decision for one or more entities – so future you will be grateful for reminders of why that position seemed like a good idea at the time.

7 Planning the Screening

[[Still have to develop this into a section]]

[[This test sentence is the first sentence written in R by Ivonne, and can be removed.]]

1) download all hits of the queries in the various databases (and through the various interfaces) as .bibtex (or .ris) files to your PC;
2) import these and deduplicate these (you can do this with a reference manager, or with the metabefor functions, which of course is what I always do because it's more transparent and efficient, and easy to re-run if you slightly change a query)
3) write the merged file to disk and send it to all screeners;
4) make sure screeners can only see title, keywords, and abstracts, and are blinded from authors, journal, and year etc;
5) let screeners indicate for each entry why it is excluded (based on a progressive list of exclusion criteria, that is based on your extraction scripts), or, for entries they cannot exclude, indicate inclusion;
6) if you have a lot of hits (thousands), usually you first screen based on title only, and only in the second round, on abstract for those entries that could not be excluded based on title;
7) after screening based on abstracts, acquire full-texts and screen those again
8) then you have your list of included sources
9) something is only excluded if all two/three/… screeners exclude it (the reason can be different; but if one screener fails to exclude, it's retained for the next step)

8 Planning the Search

8.1 Query Crafting

Running your query is the first operational step of your systematic review: it's often one of the first things you do after you publicly froze your preregistration. In that sense it's kind of exciting, but ironically normally the results you obtain will not be surprising, since you repeatedly test your query while crafting it.

8.2 Queries as logical expressions

A query is a logical expression that specifies the conditions that must be met for bibliographic records to be returned by the interfaces that you use to search the bibliographic databases (see below). You first craft this query in a conceptual form, not worrying about the syntax that you will have to use to specify your query in a way the different interfaces can parse.

The simplest queries typically bind together sets of synonyms using the logical conjunction operator (often represented by `AND`, `&`, or `&&`). Each set of synonyms binds together various terms using the logical disjunction operator (often represented by `OR`, `|`, or `||`). For example, imagine we're doing a systematic review on the determinants of ecstasy use. In that case, a simple query could be:

```
("determinant" OR "determinants" OR "correlate" OR "correlates") AND ("ecstasy" or "XTC" or "
```

This query has two terms. We could label the first "determinants", since it is intended to capture all synonyms for "determinants" (it does so badly, since I wanted to keep this example short; such lists of synonyms are normally much longer), and we could label the second "ecstasy", since its task is to match all records that contain a word for "ecstasy" (again, doing so badly to enable a brief example).

Using these two logical operators, it's also possible to build more complex queries. For example, if we would know enough about substance use to realise that the determinants of trying out (i.e. "initation" of ecstasy use) are different from what you'd find if you do a determinant study into the determinants of "using ecstasy", the second term would become more complex:

```
("determinant" OR "determinants" OR "correlate" OR "correlates") AND (("ecstasy" or "XTC" or
```

The complexity of the query you end up with is often related to the complexity or “subtlety” of your goal or research question. If you’re conducting a scoping review, where you aim to map out the literature on a specific topic, you will generally have simpler queries then when you have a specific narrow research question.

Query complexity is often also related to the richness of the literature. If the literature on a topic is very extensive, the review may become unfeasible if you use a very simple, broad query: you might obtain tens of thousands of hits without the resources to screen all of those. Similarly, if you’re surveying a smaller literature, you can afford to have a less sophisticated query. Since screening costs a lot of time, it usually pays off to spend a lot of time developing your query so that you minimize the number of irrelevant hits.

8.3 Database fields

In addition to the terms themselves, you can specify the fields you want to search. For example, you can search all text fields (the default in most interfaces if you don’t specify one or more fields), or only the title field, or the title and the abstract and the keyword fields, et cetera. Usually you will want to search the titles at the bare minimum; and unless you are confident of relatively standardized vocabulary in a field, you’ll often want to add the abstract and keyword fields. Including fields like journal name, authors, or affiliations usually doesn’t make sense, so omitting explicitly specified field names is very rare.

Sometimes interfaces will allow you to specify multiple fields in a query, for example, indicating that a search term (e.g. a set of synonyms) can occur in either the title or the abstract; but often that’s not possible, and you have to duplicate parts of your query. This can cause queries to grow exponentially, and this is one of the reasons why it is important to craft your query on the conceptual level before starting the translation into the interface languages.

8.4 Wildcard characters

The query languages used by each interface have many advanced features that you can use to build powerful queries, and it is worthwhile diving into those. In addition to logical and other operators, another category of such features is wildcard characters. For example, the asterisk (`*`) can often be used to signify “zero or more alphabetic characters”, and the question mark (`?`) can often be used to signify “zero or one alphabetic characters”. This allows you to rewrite this query fragment:

```
"behavior" OR "behaviors" OR "behavioral" OR "behaviour" OR "behaviours" OR "behavioural"
```

into this much shorter fragment:

```
"behavio?r*"
```

Because such operators differ per interface, it usually pays off to obtain a thorough understanding of the capabilities of each interface you will use before starting to craft your query (or while doing so), since you will want to create a query that's as powerful and versatile as you can, but you will have to do this within the constraints of the query languages of the interfaces you'll use.

8.5 Team Consensus and Expert Consultation

It is important to achieve consensus with the team about the query before you finalize your preregistration and then run your query "for real". If you miss important keywords, that can be a very expensive oversight to correct later on (depending on how smartly you designed your screening procedure; see below). For this reason, it is common to involve experts outside the research team to consult on the lists of synonyms and the logical structure of the query.

8.6 Databases versus Interfaces

Once you formulated your conceptual query, you can start translating it into the languages that the interfaces of the database you will use can understand. This language is generally specific to each interface. An interface is the application that performs the searches in the bibliographic databases for you and allows you to export the results in whichever format you want to use.

For example, PsycINFO is a bibliographic database maintained by the American Psychological Association. The APA keeps track of new articles that appear and adds them to PsycInfo. This database is accessible through various interfaces, and different institutions will have licenses with different interface providers. PsycInfo, for example, can be accessed through Ebsco, Ovid, and ProQuest. Ebsco, Ovid ad ProQuest use different interface languages, so the syntax and operators you have to use to formulate your query will be different.

Those interfaces are often (but not always) maintained by different organisations than those maintaining the databases. Sometimes, a database maintainer offers its own interface: PubMed is a good example of this. However, usually an interface is developed by a different organisation that then provides access to multiple databases through their interface.

This has a number of benefits. One is that once you're familiar with a given interface, you can use those skills to search multiple databases. For example, your institutions may provide access to PsycInfo, MedLine, and PsycArticles through an Ebsco interface. It also allows you to search those databases simultaneously.

It also has a number of drawbacks. First, different interfaces work differently. The available operators, the symbols representing those operators, and the syntax you have to use to build a search query therefore differs per interface. If you want to search for a word, say "meta-analysis" in an article title, sometimes you indicate this by saying `"meta-analysis" IN TI`, and sometimes by saying `TI("meta-analysis")`.

Second, the fields that exist differ per database. If you do search in multiple databases using the same interface, it is very important to clearly keep in mind which fields you search.

As a consequence of this heterogeneity in interface languages, once you crafted your conceptual query, you have to translate it into each interface's language. Depending on how many fields you want to search and on the features of each language, this can explode your query into quite unwieldy strings of characters. Make sure to document both the conceptual query and the final query you input into each database/interface combination.

Therefore, if you conduct a systematic review, it is important to always preregister both the database(s) you plan to use and the interface(s) you plan to use. In addition, it is important to document the search query you use in every interface/database combination.

8.7 "Smart" searching

When conducting a systematic review, make sure to disable all "smart" searching features of the interface(s) you use. These features expand your query by including other synonyms. However, of course, this "smart" searching algorithm is in fact dumb: it cannot understand your goal(s) and/or research question(s), and so it will simply explode your query to find many more hits, the vast majority of which will by irrelevant to your goals/questions, because after all, you crafted a well-thought-through query.

A second problem of "smart" searching is that it is not replicable, since the algorithms implemented by these interface maintainers are adjusted over time. Since you cannot encode a "smart search version" parameter in your query specification, it's not possible to solve this. As a consequence, using "smart" searching in effect renders your systematic review unsystematic: it can no longer be reproduced by other researchers — and worse, by yourself in the future.

Since systematic reviews typically take a year and often longer (see https://predicter.org/), you will often have to repeat your query towards the end, screening the additional hits, extracting entities from the additional inclusions, and re-running your analysis script. If your query was applied using "smart" searching, the results in this repeated query exectution can change unpredictably.

Therefore, never use "smart" searching in the final query you will use (and freeze in your preregistration). You *can* use it while crafting your query, to find additional sources to include, inspect the titles and abtracts for search terms you may have missed (people use the weirdest synonyms at times), and improve your conceptual query accordingly.

8.8 Query validation

Usually, you'll already have a few sources (e.g. articles, book chapters, etc) that you know you will be including in your systematic review. While testing and perfecting your query, you'll usually use these to check whether your query "works": whether it finds the articles you know it's supposed to find. If it fails to find one or more, then check whether it's supposed to find it: all bibliographic databases have a limited scope, and so the source might simply not exist in that database (easily checked by entering its title as a query). If it was supposed to be included in the hits but wasn't, then your query is missing one or more synonyms, so add those.

A quick way to check whether a given source is included in your query is by combining it with your query: basically create a "single use query" that combines the source's title (or DOI, or ISBN, or any other unique identifier for the source) with your query using the AND operator.

8.9 Exporting query hits

Once you ran your queries, you will need to download the hits: i.e. the identified bibliographic records. There is usually a set of formats available: a very common format that is generally well-supported is RIS (a format developed by "Research Information Systems"), and another good choice is BibTeX. Before deciding on the format, make sure you know how you want to conduct your screening. Ideally, you will be able to easily repeat your query later, either when you revise the manuscript to make sure your findings aren't outdated; if you updated your query because you discovered you made a mistake; or in the case of living reviews, when you want to update the review.

9 (Pre)registrations

It is best practice to (pre)register systematic reviews. This has a number of benefits, one being that others can find out that you're doing a systematic reviews, facilitating collaboration and potentially preventing double work. Conversely, before starting to plan a systematic review, you will typically want to check existing preregistration registries to make sure nobody else started the same review a year ago and is already almost done.

There is an extensive (pre)registration form for systematic reviews, the Inclusive Systematic Review Registration Form, that is available in this Google Doc, in this MetArXiv preprint, and in the `preregr` package. Using this form can be a useful help while planning your systematic review.

Part II

Execution

10 Executing the Search

10.1 Running your queries

10.2 Exporting query hits

Once you ran your queries, you will need to download the hits: i.e. the identified bibliographic records. There is usually a set of formats available: a very common format that is generally well-supported is RIS (a format developed by "Research Information Systems"), and another good choice is BibTeX. Before deciding on the format, make sure you know how you want to conduct your screening. Ideally, you will be able to easily repeat your query later, either when you revise the manuscript to make sure your findings aren't outdated; if you updated your query because you discovered you made a mistake; or in the case of living reviews, when you want to update the review.

10.3 Importing your search hits

10.4 Deduplicating your search hits

11 Executing the Screening

Screening is the process of evaluating every result of the search strategy and applying the exclusion criteria.[1]

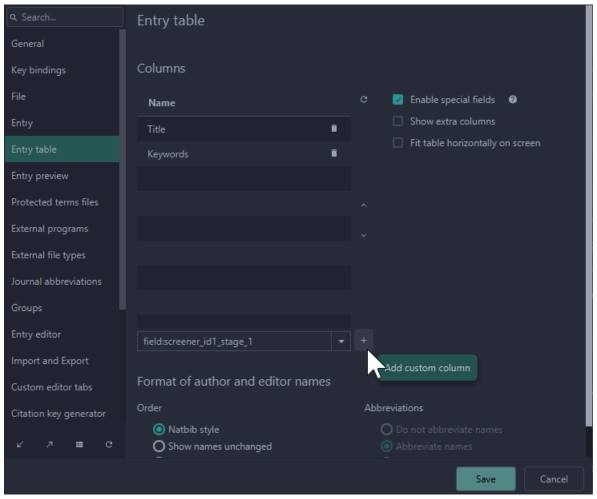
11.1 Prepare JabRef for screening

To prepare JabRef for screening, you need to mask a number of fields from the screeners, specifically any fields that may contain information that might bias the screeners. Commonly masked fields are authors, journal, and publication year. There are two places where screeners may be exposed to this information: in the entry table (the overview of all bibliographic entries in the database) and in the entry editor (where the screeners enter their screening decision).

You can change both settings in the JabRef preferences in the Options menu. To specify which fields should be visible in the entry table, open the "Entry table" section of the preferences dialog as shown in Figure 11.1. You can specify custom columns at the bottom of the overview. In this example, we add custom field `screener_id1_stage_1`, which specifies that these decisions are made by the screener with identifier `id1` in screening stage 1 (often in stage 1, only the titles are screened).

You can set the fields that screeners are exposed to in the "Custom editor tabs" section. Here, you can make tabs for each screening stage. For example, you may want to include only the title in the tab for stage 1, but also the abstract in the tab for stage 2. In additon, you'll want to include the field with the decisions for the relevant stage. For example, in if you enter a row containing "`Screening Stage 1:title;duplicate;screener_id1_stage_1`", that will add a tab called "Screening Stage 1", showing the fields "title", "duplicate", and "screener_id1_stage_1". If you add a second row containing "`Screening Stage 2:title;abstract;duplicate;screener_id1_stage_2`", that editor tab will also show the abstract, and instead of entering the decision in field `screener_id1_stage_1`, it is entered in field `screener_id1_stage_2` (i.e. the decision for screener with identifier `id` in stage 2). These rows have been entered in the example in Figure 11.2.

[1] Inclusion criteria can never override an exclusion criterion; therefore, inclusion of a soure is implicit in it not being excluded during screening.



Entry table

Columns

Name
Title
Keywords

[✓] Enable special fields
[ ] Show extra columns
[ ] Fit table horizontally on screen

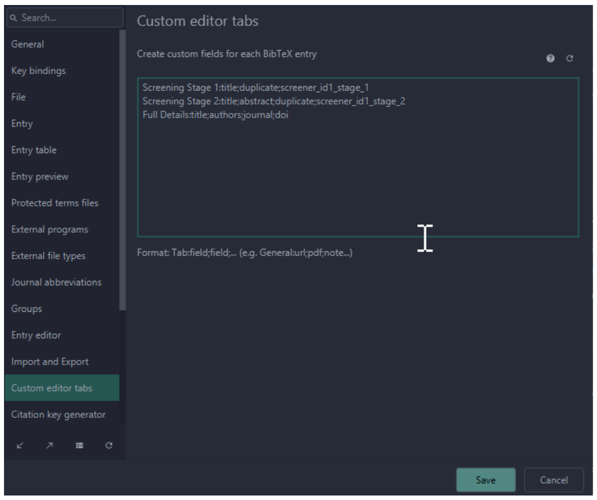
field:screener_id1_stage_1

Format of author and editor names

Order
[✓ Natbib style]
Show names unchanged

Abbreviations
Do not abbreviate names
[✓ Abbreviate names]

Figure 11.1: A screenshot of the JabRef preferences showing the ‘Entry table’ section.



Custom editor tabs

Create custom fields for each BibTeX entry

```
Screening Stage 1:title;duplicate;screener_id1_stage_1
Screening Stage 2:title;abstract;duplicate;screener_id1_stage_2
Full Details:title;authors;journal;doi
```

Format: Tab:field;field;... (e.g. General:url;pdf;note...)

Figure 11.2: A screenshot of the JabRef preferences showing the ‘Custom editor tabs’ section.

11.2 Screening in JabRef

To start screening, open the first bibliographic entry at the top of the entry table. Then, open the editor tab corresponding to the stage you're screening in.

Read fields; apply criteria; type in identifier for selected exclusion criterion or inclusion; open next entry, etc.

TODO: find shortcut key for moving through entries while staying in the 'screening' field of the entry editor.

Options -> Key bindings -> default, alt-down and alt-up. But this seems to unfocus the editor?

12 Executing the Extraction

12.1 Data management

12.2 Extracting entities

12.3 The structure of an Rxs file

Extraction occurs in Rxs files. Rxs files have the extension "`.rxs.Rmd`" (the last part "`.Rmd`", is the extension of R Markdown files; this is because Rxs files are also R Markdown files). These are plain text files that you can edit with any text editor, such as Notepad, Notepad++, TextEdit, or any other editor.

However, it is best to use RStudio, because then you can easily validate the values you extracted while you still have everything fresh in your mind.

12.3.1 The bits you can ignore

The Rxs file is structured in four sections. During extraction, only the second section matters: you can (and should) ignore the other components. This second component starts on line 4, with a comment that, by default, is the following:

```
<!--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~-->
<!--                                                                        -->
<!-- Welcome to the R Extraction Script (.rxs.Rmd file) for this source!   -->
<!--                                                                        -->
<!-- You can now start extracting. If you haven't yet studied the           -->
<!-- extractor instructions, please do so first. If you're all set, good   -->
<!-- luck!                                                                  -->
<!--                                                                        -->
<!--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~-->
```

The text in this box can be customized, so you might see something else instead. This second section ends with a similar comment, that by default has the following text:

```
<!--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~-->
<!--                                                                        -->
<!-- Well done! You are now done extracting this source. Great job!!!      -->
<!--                                                                        -->
<!-- Now, please knit the R Extraction Script into an HTML file and        -->
<!-- carefully check whether you entered everything correctly, since it    -->
<!-- will cost much less time to correct any errors, now that you still    -->
<!-- have this source in your mind, than later on when you'll have to      -->
<!-- dive into it all over.                                                 -->
<!--                                                                        -->
<!--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~-->
```

In between these two comments, you extract the values for the specified entities. Consequently, you can ignore everything above the first comment and below the second comment.

12.3.2 The sourceId block

After the first comment, the first thing you have to specify is the unique source identifier (the sourceId) for the source (e.g. an article, book, report, or other source of entities you will extract) you are extracting.

The block for the source identifier looks like this:

```
################################################################################
############################################ START: uniqueSourceIdentifier ###
################################################################################
uniqueSourceIdentifier <-
################################################################################
###
### SET UNIQUE SOURCE IDENTIFIER
###
### A unique identifier used in this systematic review to refer to this
### source
###
################################################################################

    ""

################################################################################
########################################### VALUE DESCRIPTION AND EXAMPLES ###
################################################################################
###
```

```
### A unique identifier to use in this systematic review. For sources
### with a DOI, this is the last part of the shortDOI as looked up
### through https://shortdoi.org (the part after the "10/"). For sources
### without a DOI (and so, without a shortDOI), this can be, for example,
### the QURID (Quasi-Unique Record Identifier) that was designated during
### the screening phase or which you can create with
### `metabefor::qurid()`.
###
### EXAMPLES:
###
### "g5fj"
### "qurid_7h4pksl6"
###
##############################################################################
############################################## END: uniqueSourceIdentifier ###
##############################################################################
```

As you see, there are a lot of hashes (or 'pound signs': the `#` symbol) to structure this block visually.

We will now walk through this block and inspect the bits you need to pay attention to.

12.3.2.1 The block start marker

The block starts with the block start marker. This marker indicates that this is the start of the block, which also specifies to which entity this block belongs:

```
############################################ START: uniqueSourceIdentifier ###
```

Here, you see that this entity has unique identifier "`uniqueSourceIdentifier`", which seems fitting for extracting the unique source identifiers.

12.3.2.2 The entity description sub-block

You can then ignore everything up until the sub-block with the entity's label and description:

```
##############################################################################
###
### SET UNIQUE SOURCE IDENTIFIER
###
### A unique identifier used in this systematic review to refer to this
### source
###
################################################################################
```

The entity's label is shown fully capitalized, followed by this entity's description.

12.3.2.3 The block core: where you specify the extracted value

You then see a blank line, two indented double quotes, and another blank line:

```

    ""

```

This is the core of this block. It is where the entity value that you extract is specified (in between the double quotes). The double quotes are the default value for this entity. In this case, they represent an empty text string.

> **Note**
>
> Note that for computers, literal text strings always have to be in between either a pair of double quotes (`"like this"`) or a pair of single quotes (`'like this'`). Double quotes usually make the most sense, because single quotes also serve other functions as apostrophes, so when you copy-paste a text, especially in Engelish, it's likely to contain one or more single quotes (e.g. in the word "`it's`"). If that text string is then delimited by single quotes, you will cause an error, because as far as the computer knows, you unintentionally stop the text string specification as soon as the first "`'`" in the text string is encountered.

12.3.2.4 The value description sub-block

This position, where you specify the value for this entity, is followed by another sub-block:

```
################################################################################
########################################### VALUE DESCRIPTION AND EXAMPLES ###
################################################################################
###
### A unique identifier to use in this systematic review. For sources
### with a DOI, this is the last part of the shortDOI as looked up
### through https://shortdoi.org (the part after the "10/"). For sources
### without a DOI (and so, without a shortDOI), this can be, for example,
### the QURID (Quasi-Unique Record Identifier) that was designated during
### the screening phase or which you can create with
### `metabefor::qurid()`.
###
### EXAMPLES:
###
### "g5fj"
### "qurid_7h4pksl6"
###
##############################################################################
```

This sub-block is marked "`VALUE DESCRIPTION AND EXAMPLES`". Unlike the entity label and description that we saw above, this sub-block tells you *what kind of value* has to be extracted for this entity. So, whereas the first sub-block tells you what you need to specify here (e.g. what to look for in the PDF of a source), this second sub-block tells you how you need to specify what you extracted.

For example, some entities always have to be numbers (e.g., number of participants in a study, or a correlation coefficient). Some numbers always have to be whole numbers, without decimals (e.g., number of participants in a study), whereas others can contain decimals (e.g., a correlation coefficient). Sometimes you need to extract a text string. Sometimes you can extract a so-called vector containing multiple text strings or numbers.

This description below the block core (where you specify the value you extracted for this entity) tells you how to specify that value given what was decided during the planning of this systematic review.

Note

You specify a vector using "`c()`". For example, to extract to numbers, say `1` and `2`, you would specify "`c(1, 2)`". This "`c()`" stands for "combine", because it lets you combine two or more values into one vector.

12.3.2.5 The block end marker

Finally, this block ends with the block end marker:

```
##############################################################################
############################################### END: uniqueSourceIdentifier ###
##############################################################################
```

These are basically the exact same lines as the block start marker, except that the entity identifier for this block is now preceded by the word "`END`", instead of by the word "`START`".

This block end marker closes the block for this entity. In this case, this means the block for the source identifier is done.

12.3.2.6 The value to specify here

The value you specify as "`uniqueSourceIdentifier`" is described in the value description sub-block (the sub-block marked with "`VALUE DESCRIPTION AND EXAMPLES`").

If you don't have it yet, you'll have to get the ShortDOI for this source, assuming the source has a DOI. A ShortDOI is a unique brief unique identifier for any object that has a Digital Object Identifier (i.e. a DOI). You can find the ShortDOI that corresponds to a given DOI at https://shortdoi.org.

If you're extracting a source that doesn't have a DOI, normally, during screening every screened entry will have received a QURID, a Quasi-Unique Record Identifier. You can then copy that from the extraction spreadsheet, which is where you probably also have to copy-paste the ShortDOI if that's what extracted as the value for "`uniqueSourceIdentifier`" (refer to your extraction instructions for the details).

When you're done, the block core should look something like this:

```
     "gqw5jr"
```

Note that you shouldn't include the full ShortDOI, but instead omit the "`10/`" that it starts with, so you only specify letters and digits.

12.3.3 The extractorId block

Then, you move on to the second block: the extractorId block. This extractor identifier is the identifier for you, so that later on in the project, it's still clear who extracted this source.

With your team, you will agree on identifiers for every extractor. Note that because the Rxs files will be made public, these extractor identifiers will also become public, so you may want to avoid using your name (on the other hand, this could also be a reason to deliberately use your name: just think about what you would prefer).

The extractor identifier block looks like this:

```
##############################################################################
############################################### START: extractorIdentifier ###
##############################################################################
extractorIdentifier <-
```

```
################################################################################
###
### SPECIFY YOUR EXTRACTOR IDENTIFIER
###
### An identifier unique to every extractor
###
################################################################################

    ""

################################################################################
########################################### VALUE DESCRIPTION AND EXAMPLES ###
################################################################################
###
### Identifiers can only consist of (lower or uppercase) Latin letters
### [a-zA-Z], Arabic numerals [0-9], and underscores [_], and always have
### to start with a letter.
###
### EXAMPLES:
###
### "extractor_1"
### "Alex"
###
################################################################################
################################################# END: extractorIdentifier ###
################################################################################
```

You should now recognize the block start marker (the bit saying "`START: extractorIdentifier`") and the block end marker (the bit saying "`END: extractorIdentifier`"), as well as the entity description sub-block (with label "`SPECIFY YOUR EXTRACTOR IDENTIFIER`" and description "`An identifier unique to every extractor`"), the block core (the two double quotes, "`""`"), and the value description sub-block (marked with "`VALUE DESCRIPTION AND EXAMPLES`" and then proceeding to describe the constraints that an identifier must satisfy).

In this case, you type or copy-paste your personal extractor identifier in between the double quotes in the block core. When you're done, the block core should look something like this:

```
    "myExtractorId"
```

12.3.4 Starting the actual extraction

Now that you've specified the relevant metadata for this extraction (because you haven't actually really extracted anything from the source yet…), you can start with the actual entities to extract.

This part starts with the following lines:

```
##############################################################################
##################################################### START: source (ROOT) ###
##############################################################################
rxsObject <- data.tree::Node$new('source');
currentEntity <- rxsObject;
##############################################################################
```

You can ignore this: it just shows when the metadata (the source identifier and extractor identifier) end and the real entities to extract start.

This start also has a corresponding block all the way at the bottom, just before the closing comment:

```
##############################################################################
####################################################### END: source (ROOT) ###
##############################################################################
class(rxsObject) <- c('rxs', 'rxsObject', class(rxsObject));
rxsObject$rxsMetadata <- list(rxsVersion='0.3.0', moduleId=NULL, id=uniqueSourceIdentifier
##############################################################################
##############################################################################
##############################################################################
```

In between these two blocks, you specify the values for the actual entities.

In principle, this process is relatively simple: you just scroll further down in the Rxs file, and for every entity block, you specify in the block core whatever is explained for that entity and the required value.

There are three more patterns that you'll likely encounter, so let's look at those first.

12.3.5 Entity containers

Entities are usually organized hierarchically (in a nested, tree-like structure). This is useful because the point of using Rxs files is that it's easy to combine extracted entities from multiple files for the same source. This way, many people can easily collaborate on the same database of machine-readable literature. You can even collaborate with "future you": it's easy to first do a relatively superficial extraction pass, and later on specify more detailed entities in another Rxs specification, extract those, and then combine all data in one database.

Entity containers are entities that are themselves not extractable, but that just exist to contain other entities. Common entity containers are, for example, “General”, “Methods”, and “Results”. Every time you encounter a container entity, the blocks of hashes indent by two spaces.

This indenting starts immediately after the “source root” container has opened:

```
###############################################################################
###################################################### START: source (ROOT) ###
###############################################################################
rxsObject <- data.tree::Node$new('source');
currentEntity <- rxsObject;
###############################################################################

  #############################################################################
  ########################################################## START: general ###
  #############################################################################
  currentEntity <- currentEntity$AddChild('general');
  #############################################################################
  ###
  ### GENERAL
  ###
  ### General information about the article
  ###
  #############################################################################
```

In this example, the source root contains an entity container with entity identifier “general”. This entity container has label “`GENERAL`” and description “`General information about the article`”.

This entity container does not itself contain a corresponding value: instead, the Rxs file indents again and an entity block (for the entity with identifier “`qurid`”) is shown:

```
    ###########################################################################
    ########################################################## START: qurid ###
    ###########################################################################
    currentEntity <- currentEntity$AddChild('qurid');
    currentEntity[['value']] <-
    ###########################################################################
    ###
    ### QURID
    ###
    ### Quasi-Unique Record Identifier. We will use this to
    ### supplement this information with information from the
    ### bibliographic databases (i.e. the screening database).
    ###
    ##########################################################################

        NA

    ##########################################################################
    ####################################### VALUE DESCRIPTION AND EXAMPLES ###
    ##########################################################################
    ###
    ### A single character value
    ###
    ### EXAMPLES:
    ###
    ### "Example"
    ### "Another example"
    ###
    ##########################################################################
    currentEntity[['validation']] <- expression(is.na(VALUE) || (is.character(VALUE) && le
    currentEntity <- currentEntity$parent;
    ##########################################################################
    ########################################################### END: qurid ###
    ##########################################################################
```

Further down, this entity container ends with a block end marker for this entity container (which had "`general`" as entity identifier):

```
    ############################################################################
    ############################################################################
    currentEntity <- currentEntity$parent;
    ############################################################################
    ########################################################### END: general ###
    ############################################################################
```

12.3.6 Clustering entities

Often, you will want to extract multiple closely related values; or you want to extract something that can have many different forms in a source, for example an effect size, where you need to know what you're extracting exactly. In those situations, you usually use "clustering entities" or "list entities".

The look like an entity where you don't extract just one value, but multiple labelled values. An example is shown below:

```
############################################################################
############################################# START: authors (REPEATING) ###
############################################################################
currentEntity <- currentEntity$AddChild('authors__1__');
currentEntity[['value']] <-
############################################################################
###
### AUTHORS
###
### Information about each author. Note that authors are repeating;
### therefore, copy the list below multiple times if there are
### multiple authors. Fill it out in the order of authorship.
###
############################################################################

    list(authorId = NA,             ### Author identifier: A unique identifier for this a
         authorName = NA,           ### Author name: The full name of this author. [Examp
         authorORCID = NA,          ### Author ORCID: The author's ORCID, if available or
         authorAffiliation = NA);   ### Author affiliation: The author's affiliations as 

############################################################################
currentEntity[['validation']] <- list(`authorName` = expression(is.na(VALUE) || (is.char
                                      `authorORCID` = expression(is.na(VALUE) || (is.cha
                                      `authorAffiliation` = expression(is.na(VALUE) || (
currentEntity$name <- metabefor::nodeName(currentEntity$value[[1]], "authors__1__");
currentEntity <- currentEntity$parent;
############################################################################
############################################### END: authors (REPEATING) ###
############################################################################
```

Let's take a closer look at the block core in this entity block:

```
    list(authorId = NA,             ### Author identifier: A unique identifier for this a
         authorName = NA,           ### Author name: The full name of this author. [Examp
         authorORCID = NA,          ### Author ORCID: The author's ORCID, if available or
         authorAffiliation = NA);   ### Author affiliation: The author's affiliations as 
```

We see that this block core consists of "`list()`", in between those parentheses, there is a list of three entity identifiers, each followed by an equals sign, "`NA`", and then, after some spaces, three hashes (`###`), the entity labels and description, and between square brackets ("`[`" and "`]`"), examples and value descriptions.

The default values for these three entities is "`NA`", which stands for "Not Applicable" (but you can read it as "Not Extracted Yet". When you extract these three entities, you still have to supply the double quotes yourself. Once you extracted this clustering entity (or list entity), this core block might look like this:

```
        list(authorId = "viechtbauer",            ### Author identifier: A unique identifier
             authorName = "Wolfgang Viechtbauer",          ### Author name: The full name of
             authorORCID = "0000-0003-3463-4063",         ### Author ORCID: The author's ORC
             authorAffiliation = "02jz4aj89");  ### Author affiliation: The author's affilia
```

Clustering entities (or list entities, whatever you prefer to call them) are in the end just convenient ways to more quickly extract closely related entities.

12.3.7 Repeating entities

The final common pattern you may encounter are repeating entities. Repeating entities are used in situations where during the planning of the systematic review, you cannot be sure how often a given entity will occur in a source. This can be the case, for example, with samples in a study; or with authors; or with countries where data were collected; or many other entities.

The solution is relatively simple: you just copy-paste the entity block for a repeating entity. If you look back, you saw that the entity block for entity "`authors`" was repeating. You can see this by inspecting the block start marker and the block end marker.

The block start marker was:

```
    ###########################################################################
    ############################################## START: authors (REPEATING) ###
    ###########################################################################
```

The block end marker was:

```
    ###########################################################################
    ################################################ END: authors (REPEATING) ###
    ###########################################################################
```

The text "`(repeating)`" after the entity identifier tells you that this is a repeating entity.

Therefore, if this source has multiple authors (and most sources do), you copy the entire block (including the block start marker and the block end marker) and you paste it right below (with an empty line in between so you can easily spot where one entity block ends and the next one starts.

If you forget to specify a valid identifier for the first entity in a repeating list entity, `{metabefor}` will throw an error when you try to "knit" or "render" the Rxs file (using CTRL-SHIFT-K).

For example, in this case it would say something like:

```
Quitting from lines 15-883 [rxs-extraction-chunk] (extractionScriptTemplate.rxs.Rmd)

Error in `metabefor::nodeName()`:
!
---------- metabefor error, please read carefully:

As an identifier for this entity (with temporary name
'authors__1__'), you specified `NA` (you probably forgot to
specify an identifier). Please change it to a valid entity
identifier!

Identifiers can only consist of (lower or uppercase) Latin
letters [a-zA-Z], Arabic numerals [0-9], and underscores
[_], and always have to start with a letter (as a regular
expression: ^[a-zA-Z][a-zA-Z0-9_]*$).

----------

Backtrace:
 1. metabefor::nodeName(currentEntity$value[[1]], "authors__1__")
Execution halted
```

This error shows up in the R console in RStudio, by default located in the bottom-left corner of RStudio.

In this case, fix that identifier and try again.

You then specify the values for the second occurrence of this entity (in this case, for the second author). This way, you are able to extract as many repetitions of this entity as the source may contain.

If you're done, hit CTRL-SHIFT-K again to "knit" or "render" the Rxs file. If everything goes well, you should see something similar to what you see at https://metabefor.opens.science/articles/validation.html.

In that case, check whether every entity validates succesfully. If so, check the table at the bottom and make sure that all values as they show up there are correct. These are the values as they are imported, so if they show up correctly here, you know that everything went well. If not, correct whatever is not going well.

Congratulations - you extracted a source, made a little bit of the literature machine-readable, and so contributed to scientific progress and a better world!

12.4 Validation of extracted entities

12.5 Contact with authors

13 Executing the Synthesis

This chapter still has to be written.

Part III

Operations

14 Ops for the search

15 Ops for the screening

15.1 First time: preparing a PC

To prepare your PC for screening, you need to do two things. First, download JabRef. Second, configure JabRef.

15.1.1 Downloading and installing JabRef

To download JabRef, visit https://jabref.org and click the “Download” hyperlink or scroll down to the Download section. Then download and install JabRef.

15.1.2 Configuring JabRef

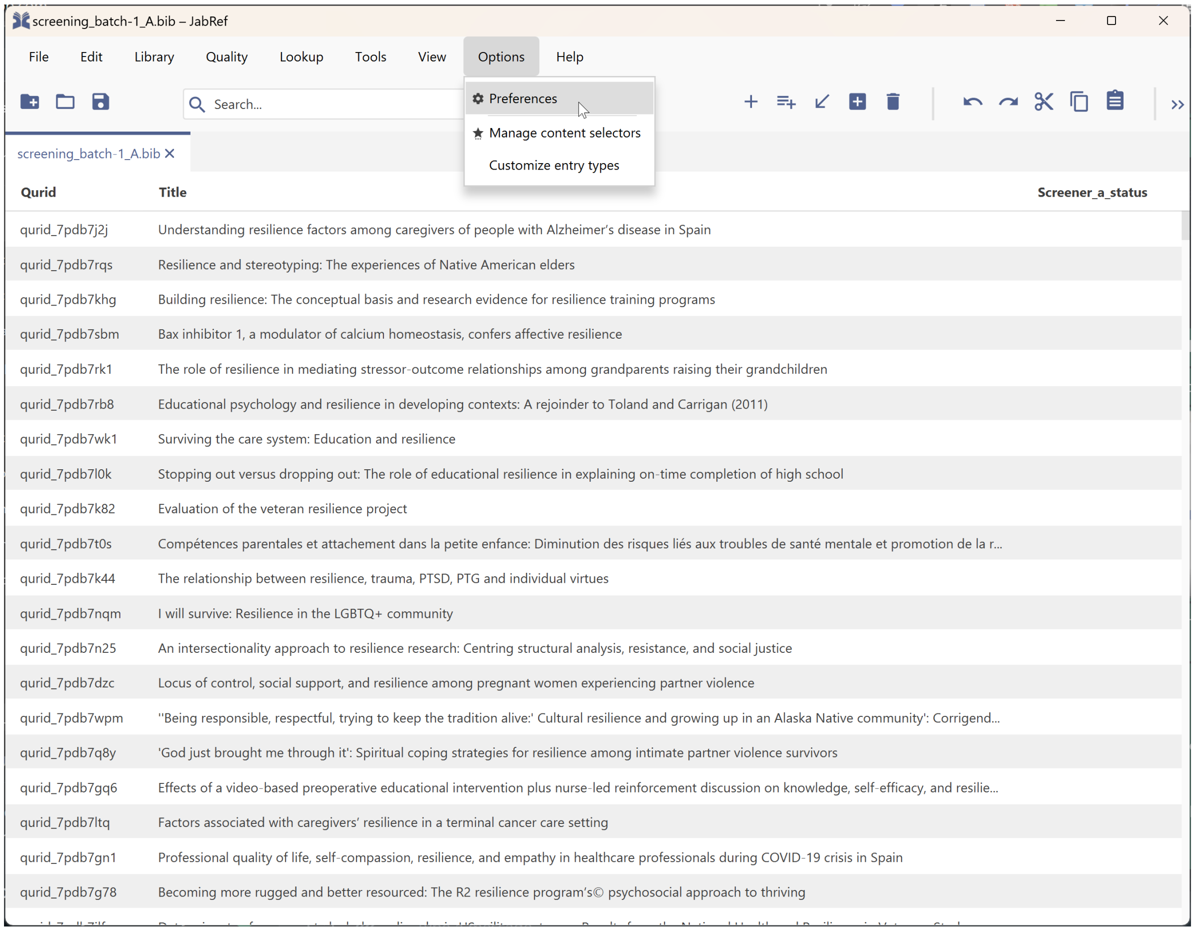
Then, you need to configure JabRef. JabRef is not created for screeening; it is a reference manager, like Zotero (or the ‘Closed Science’ alternatives such as EndNote, Mendeley, or RefManager). However, it is exceptionally well suited for screening because of two reasons. First, it is very customizable - so customizable that you can hide arbitrary information, making it possible to mask screeners from, for example, authors, journal name or publication year. Second, it uses the BibTeX file format and supports arbitrary field names. This makes it easy to create dedicated fields where the screening decisions can be stored.

The configuration of JabRef consists of a number of actions to optimize JabRef for screening comfort, efficiency, and integrity (i.e., masking the screeners from potentially biasing information). The following screenshots were taken with JabRef version 5.9[1], so if you have another version, your interface might look slightly different.

15.1.2.1 Configuring JabRef: opening the Preferences dialog

To open the options, click the Options menu and select the Preferences option as shown in Figure 15.1.

[1]Specifically, JabRef 5.9–2023-01-08–76253f1a7 on Windows 11 10.0 amd64 and running Java 19.0.1 and JavaFX 19+11.



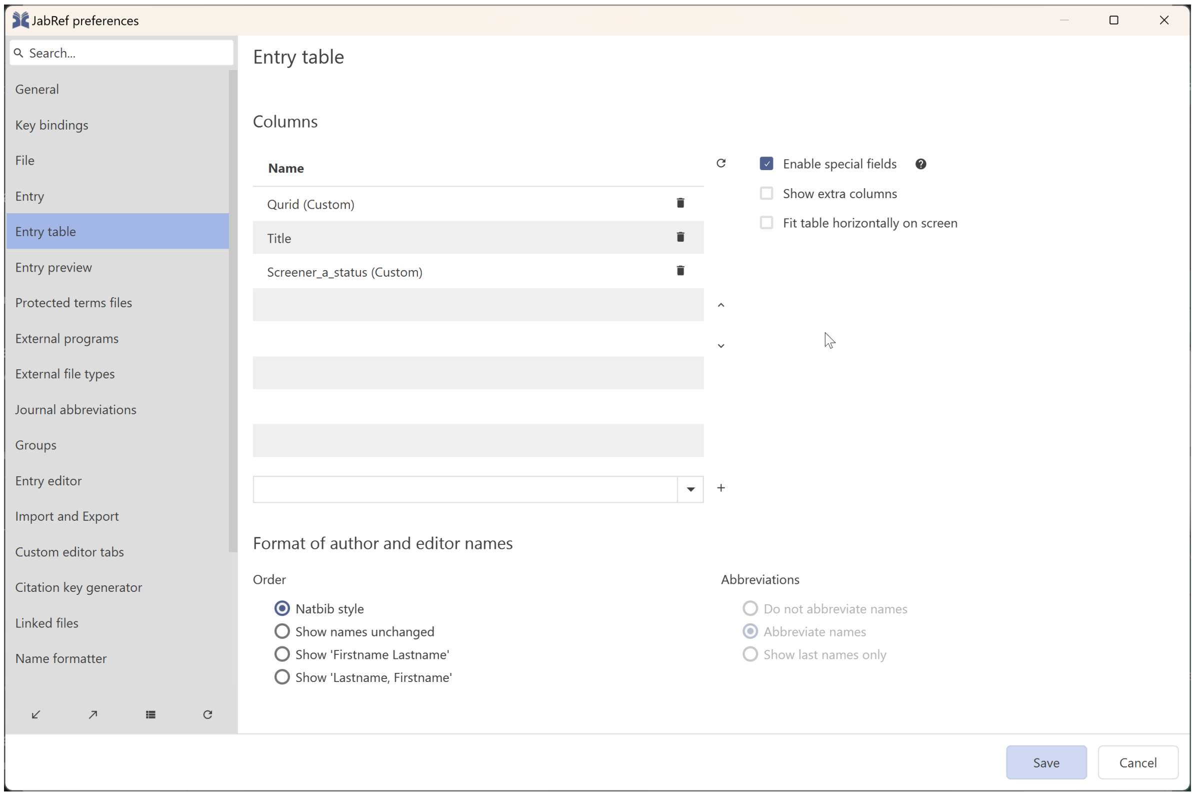
Qurid	Title	Screener_a_status
qurid_7pdb7j2j	Understanding resilience factors among caregivers of people with Alzheimer's disease in Spain	
qurid_7pdb7rqs	Resilience and stereotyping: The experiences of Native American elders	
qurid_7pdb7khg	Building resilience: The conceptual basis and research evidence for resilience training programs	
qurid_7pdb7sbm	Bax inhibitor 1, a modulator of calcium homeostasis, confers affective resilience	
qurid_7pdb7rk1	The role of resilience in mediating stressor-outcome relationships among grandparents raising their grandchildren	
qurid_7pdb7rb8	Educational psychology and resilience in developing contexts: A rejoinder to Toland and Carrigan (2011)	
qurid_7pdb7wk1	Surviving the care system: Education and resilience	
qurid_7pdb7l0k	Stopping out versus dropping out: The role of educational resilience in explaining on-time completion of high school	
qurid_7pdb7k82	Evaluation of the veteran resilience project	
qurid_7pdb7t0s	Compétences parentales et attachement dans la petite enfance: Diminution des risques liés aux troubles de santé mentale et promotion de la r...	
qurid_7pdb7k44	The relationship between resilience, trauma, PTSD, PTG and individual virtues	
qurid_7pdb7nqm	I will survive: Resilience in the LGBTQ+ community	
qurid_7pdb7n25	An intersectionality approach to resilience research: Centring structural analysis, resistance, and social justice	
qurid_7pdb7dzc	Locus of control, social support, and resilience among pregnant women experiencing partner violence	
qurid_7pdb7wpm	''Being responsible, respectful, trying to keep the tradition alive:' Cultural resilience and growing up in an Alaska Native community': Corrigend...	
qurid_7pdb7q8y	'God just brought me through it': Spiritual coping strategies for resilience among intimate partner violence survivors	
qurid_7pdb7gq6	Effects of a video-based preoperative educational intervention plus nurse-led reinforcement discussion on knowledge, self-efficacy, and resilie...	
qurid_7pdb7ltq	Factors associated with caregivers' resilience in a terminal cancer care setting	
qurid_7pdb7gn1	Professional quality of life, self-compassion, resilience, and empathy in healthcare professionals during COVID-19 crisis in Spain	
qurid_7pdb7g78	Becoming more rugged and better resourced: The R2 resilience program's© psychosocial approach to thriving	

Figure 15.1: The JabRef interface with the Options menu opened and the Preferences option highlighted

15.1.2.2 Configuring JabRef: the Entry Table columns

Open the section for the Entry Table (see Figure 15.2). Below the table labelled "Columns", add two new fields. The first is labelled "qurid" (assuming that's where the quasi-unique record identifiers, the QURIDs, are stored; you create these with `metabefor::generate_qurids()`). The second is the name of the field you will use for the screening. This field is set when calling `metabefor::write_screenerPackage()`. If you yourself do not manage this part of the project, the person(s) who do probably informed you of this field name (it will generally contain the identifier for every screener, and may also contain an identifier for the screening round, if screening occurs in multiple rounds, and finally it may contain an identifier for the screening batch, if screening occurs in multiple batches). In Figure 15.2, this field is "Screener_a_status".

Once you added both of these fields, remove all other fields except for "Title", so you remain with only three fields (as shown in Figure 15.2).



Entry table

Columns

Name
Qurid (Custom)
Title
Screener_a_status (Custom)

[✓] Enable special fields
[ ] Show extra columns
[ ] Fit table horizontally on screen

Format of author and editor names

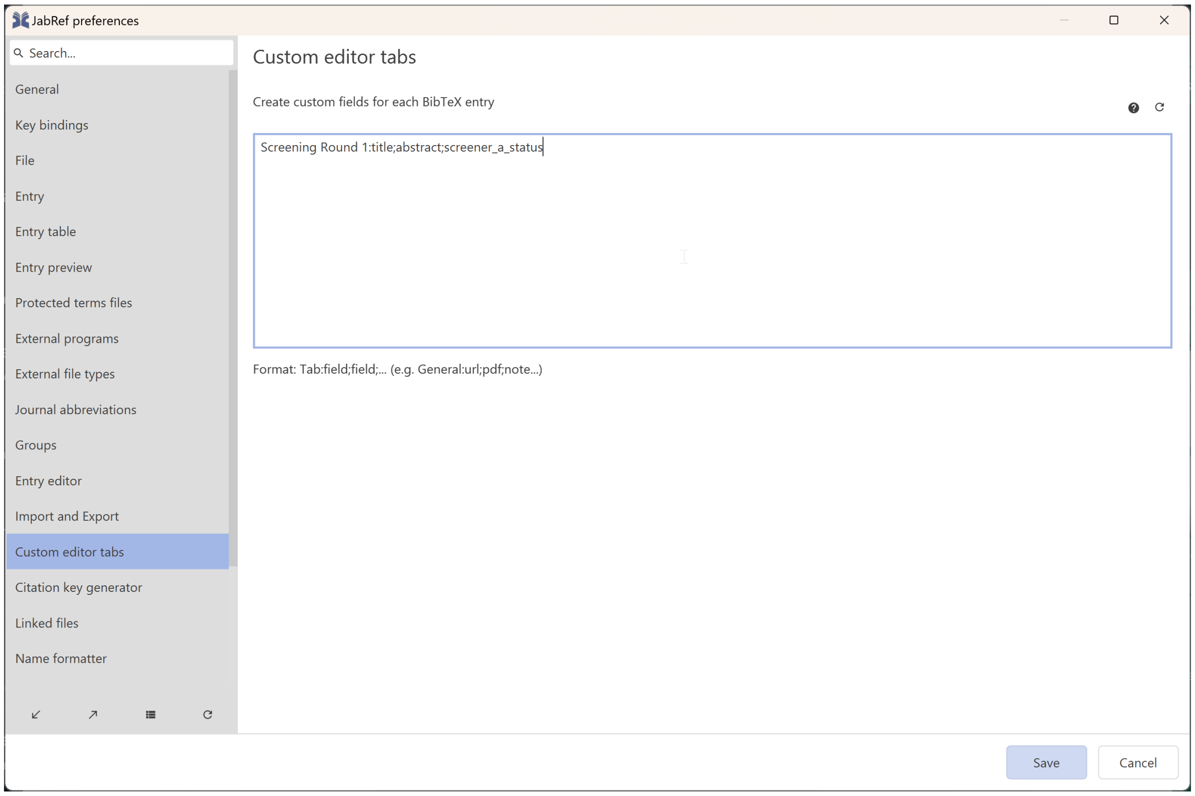
Order
[✓ Natbib style]
Show names unchanged
Show 'Firstname Lastname'
Show 'Lastname, Firstname'

Abbreviations
Do not abbreviate names
[✓ Abbreviate names]
Show last names only

Figure 15.2: The JabRef interface showing the Entry Table customization dialog

15.1.2.3 Configuring JabRef: the Entry Editor tab

Open the section for the Entry Table (see Figure 15.3). In the large text field, every line represents one custom entry editor tab. Add a line with the name of the tab (e.g. "Screening", or, as in the example in Figure 15.3, "Screening Round 1"), immediately followed by a colon (":"), and then followed by the fields to show separated by semicolons (";"). Specify the "title" and "abstract" fields as well as the field you use to store the screening decisions. This is the same field that you added to the Entry Table in the previous step (in our example in Figure 15.2 and here in **?@fig-ops-screening-entry-editorm** this is "screener_a_status").



Custom editor tabs

Create custom fields for each BibTeX entry

```
Screening Round 1:title;abstract;screener_a_status
```

Format: Tab:field;field;... (e.g. General:url;pdf;note...)

Figure 15.3: The JabRef interface showing the Entry Editor Tabs dialog

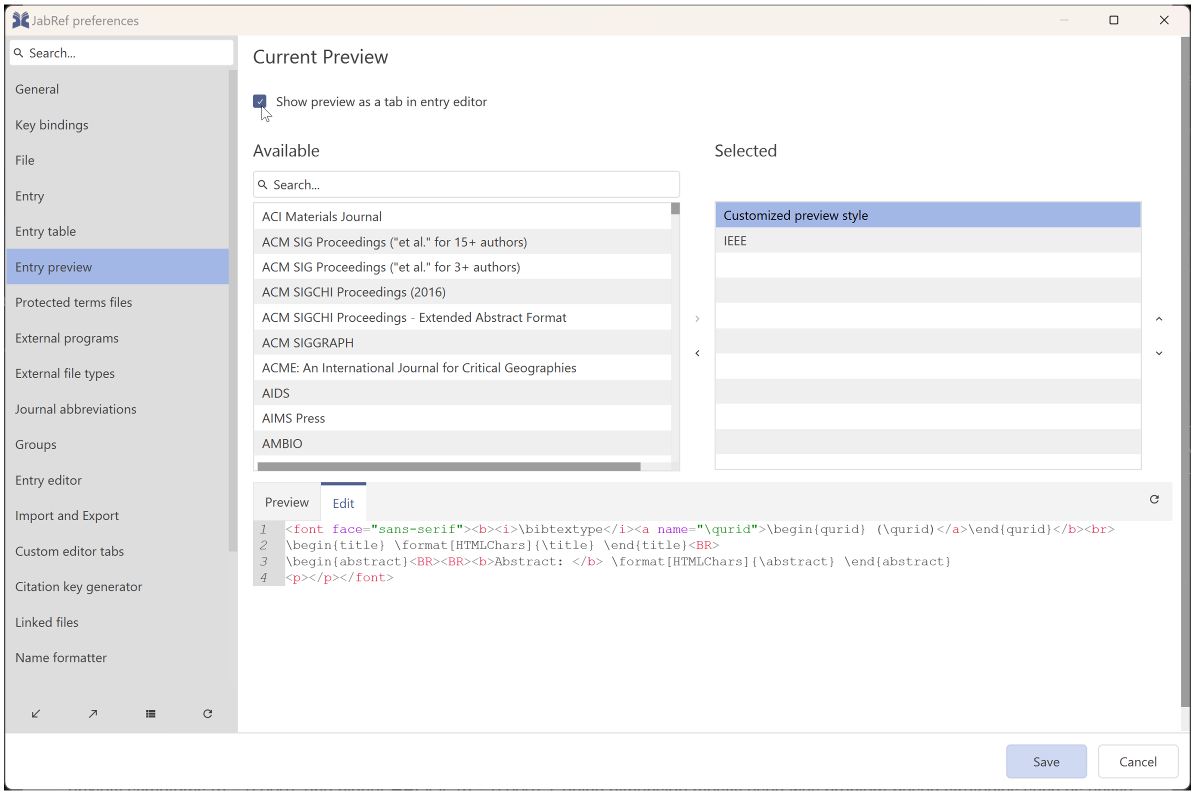
15.1.2.4 Configuring JabRef: the Entry Preview tab

Open the section for the Entry Preview (see Figure 15.4). Select a preview style in the box on the left and click the "Edit" tab. Then, copy-paste the following text:

```
<font face="sans-serif">
<b><i>\bibtextype</i><a name="\qurid">\begin{qurid} (\qurid)</a>\end{qurid}</b><br>
\begin{title} \format[HTMLChars]{\title} \end{title}<br>
\begin{abstract}<BR><BR><b>Abstract: </b> \format[HTMLChars]{\abstract} \end{abstract}
<p></p></font>
```

This will ensure the entry preview also will not disclose information that the screeners should be masked from (e.g. authors, journal, or publication year).

Instead of doing this, you can also achieve the same result by checking the checkbox for "Show preview as a tab in entry editor" if you want. This will mean you no longer see the entry preview to the right of the entry editor, but it's in its own tab instead (see Figure 15.4).



[✓] Show preview as a tab in entry editor

Available:
ACI Materials Journal
ACM SIG Proceedings ("et al." for 15+ authors)
ACM SIG Proceedings ("et al." for 3+ authors)
ACM SIGCHI Proceedings (2016)
ACM SIGCHI Proceedings - Extended Abstract Format
ACM SIGGRAPH
ACME: An International Journal for Critical Geographies
AIDS
AIMS Press
AMBIO

Selected:
Customized preview style
IEEE

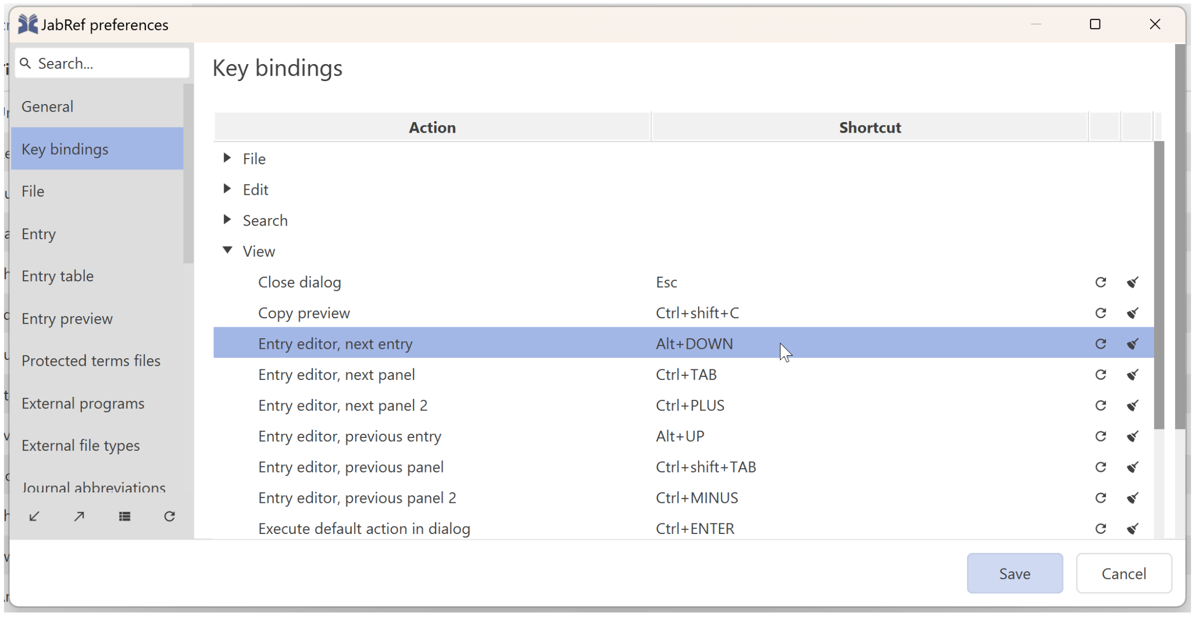
```
<font face="sans-serif"><b><i>\bibtextype</i><a name="\qurid">\begin{qurid} (\qurid)</a>\end{qurid}</b><br>
\begin{title} \format[HTMLChars]{\title} \end{title}<BR>
\begin{abstract}<BR><BR><b>Abstract: </b> \format[HTMLChars]{\abstract} \end{abstract}
<p></p></font>
```

Figure 15.4: The JabRef interface showing the Entry Preview customization dialog

15.1.2.5 Configuring JabRef: Shortcut keys

The last customization is setting the shortcut keys to easily cycle through the hits to screen. You set these in the Key Bindings section (see Figure 15.5).

Expand the actions clustered in the “View” cluster and set the shortcut key you want for “Entry editor, next entry” and “Entry editor, previous entry”. This will allow you to cycle through the entries without having to use the mouse, which is considerably more efficient.



Key bindings

Action	Shortcut
▸ File	
▸ Edit	
▸ Search	
▾ View	
Close dialog	Esc
Copy preview	Ctrl+shift+C
Entry editor, next entry	Alt+DOWN
Entry editor, next panel	Ctrl+TAB
Entry editor, next panel 2	Ctrl+PLUS
Entry editor, previous entry	Alt+UP
Entry editor, previous panel	Ctrl+shift+TAB
Entry editor, previous panel 2	Ctrl+MINUS
Execute default action in dialog	Ctrl+ENTER

Figure 15.5: The JabRef interface showing the Key Bindings customization dialog

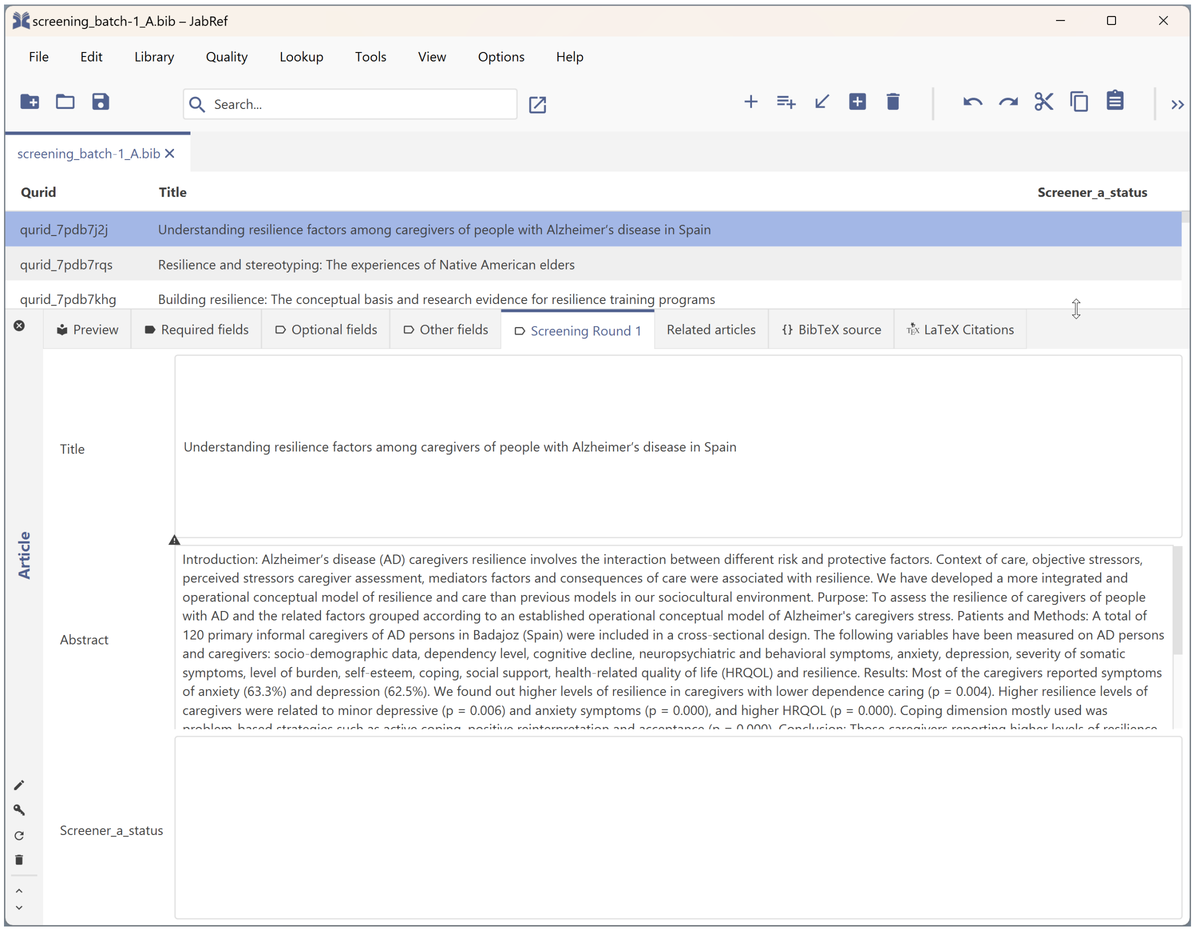
15.1.2.6 Configuring JabRef: separator dragging

Finally, drag the horizontal separator between the entry table and the entry editor up, so that the entry editor becomes larger (see Figure 15.6).

15.2 In every screening session

Every time you do a screening session, you do the following.

1. Open JabRef.
2. In JabRef, open the `.Bib` file containing the hits.
3. Locate the entry where you want to start screening.
4. Double click it to open it. Potentially adjust the horizontal separator between the entry table (showing all entries) and the entry editor so that you can read as much of the abstract as possible.



Qurid	Title	Screener_a_status
qurid_7pdb7j2j	Understanding resilience factors among caregivers of people with Alzheimer's disease in Spain	
qurid_7pdb7rqs	Resilience and stereotyping: The experiences of Native American elders	
qurid_7pdb7khg	Building resilience: The conceptual basis and research evidence for resilience training programs	

Title	Understanding resilience factors among caregivers of people with Alzheimer's disease in Spain
Abstract	Introduction: Alzheimer's disease (AD) caregivers resilience involves the interaction between different risk and protective factors. Context of care, objective stressors, perceived stressors caregiver assessment, mediators factors and consequences of care were associated with resilience. We have developed a more integrated and operational conceptual model of resilience and care than previous models in our sociocultural environment. Purpose: To assess the resilience of caregivers of people with AD and the related factors grouped according to an established operational conceptual model of Alzheimer's caregivers stress. Patients and Methods: A total of 120 primary informal caregivers of AD persons in Badajoz (Spain) were included in a cross-sectional design. The following variables have been measured on AD persons and caregivers: socio-demographic data, dependency level, cognitive decline, neuropsychiatric and behavioral symptoms, anxiety, depression, severity of somatic symptoms, level of burden, self-esteem, coping, social support, health-related quality of life (HRQOL) and resilience. Results: Most of the caregivers reported symptoms of anxiety (63.3%) and depression (62.5%). We found out higher levels of resilience in caregivers with lower dependence caring (p = 0.004). Higher resilience levels of caregivers were related to minor depressive (p = 0.006) and anxiety symptoms (p = 0.000), and higher HRQOL (p = 0.000). Coping dimension mostly used was
Screener_a_status	

Figure 15.6: Dragging and dropping a panel separator in JabRef

5. Apply the first exclusion criterion in the list of progressive exclusion criteria. If you are confident, based on the title and the abstract, that the description of that exclusion criterion applies to this hit (called "entry" in JabRef), type the corresponding code in the screener field. If not, progress to the next exclusion criterion.
6. Apply the next exclusion criterion in the list of progressive exclusion criteria. If you are confident, based on the title and the abstract, that the description of that exclusion criterion applies to this hit (called "entry" in JabRef), type the corresponding code in the screener field. If not, progress to the next exclusion criterion.
7. Repeat 6 until you have evaluated all exclusion criteria. If you cannot confidently say that one of the exclusion criteria applies to this hit/entry/source, it is included (i.e. will be retained in this phase). Enter the corresponding code into the screening field (e.g. "IN" or "Incl" or "Included").
8. Move to the next entry/hit/source by clicking it in the entry table at the top, or by using the key combination you configured for "Entry editor, next entry" when preparing your computer for screening.
9. If you have screened all entries/hits/sources (i.e. you have typed something in the screening field of all of them), you are done. Save the `.bib` file, close JabRef, and store the `.bib` file in whichever location is listed in your project's data management plan. Sometimes, this is handled by somebody else - in that case, send the `.bib` file to them.
10. If you are not done, but taking a break, depending on what you agreed on, just save the `.bib` file and JabRef and you're done; or store the intermediate version in the location in your project's data management plan or send it to whoever handles the project's data management (as extra security/redundancy).

Well done - you've now created an open, machine readable list of your screening decisions that can then be pooled with other batches, screening decisions from other screeners, or even other projects.

16 Ops for the extraction

16.1 First time: preparing a PC

16.1.1 Installing the software

Whether you need to install any software for the extraction is mostly a matter of personal preference. You will need to validate every extraction script to make sure you didn't accidently make a typo (e.g. forgot a quote) or extracted an entity in the wrong format.

One option is to do this using EVA (the Extraction Validation App), at https://opens.science/apps/eva. In that case, the only software you need is an application to edit plain text files. You can use whatever came with your operating system (e.g. Notepad in Windows, Text Edit in MacOS, or GNOME Text Editor in Ubuntu). You can also install a more powerful text editor, like Notepad++.

Alternatively, you can use RStudio and R. This has two advantages over the approach discussed in the last paragraph. First, you have the benefit of RStudio's syntax coloring. This is very helpful to help you avoid accidently omitting a quote or a parenthesis. In addition, you can then directly validate your extraction by 'knitting' or 'rendering' the R Extraction Script in R. If you want to use R and RStudio, you have to install these programs if they are not present yet:

1. R: link to download R for Windows
2. RStudio link to download RStudio
3. Git link to download Git

For all three programs, you can accept the default options in the installation, but for Git, you may want to select that you use Notepad (or Notepad++ if you have it) as the default editor instead of Vim; and you may want to select "main" as default branch name instead of "master".

16.1.2 Installing the R packages

If you want to use R, you will also need to install a number of R packages. Once you installed R and RStudio, start RStudio and then run these commands by copying them and pasting them in the console (the bottom-left panel in RStudio):

```
install.packages(c('remotes', 'here', 'markdown', 'commonmark'),
                 repos='http://cran.rstudio.com');
install.packages(c('ufs', 'ggplot2', 'DiagrammeR', 'DiagrammeRsvg'),
                 repos='http://cran.rstudio.com');
install.packages(c('yum', 'synthesisr', 'preregr', 'rock'),
                 repos='http://cran.rstudio.com');
ufs::quietGitLabUpdate("r-packages/metabefor", quiet = FALSE);
```

The last of these installs the `{metabefor}` package from its Git repository. Once `{metabefor}` is on CRAN, you can install from there using `install.packages()`.

16.1.3 Cloning the repository

Cloning means that you copy a Git repository to your local PC. This will download all files in the project, along with the metadata needed to pull from and push to the server (e.g., a GitLab or Codeberg server).

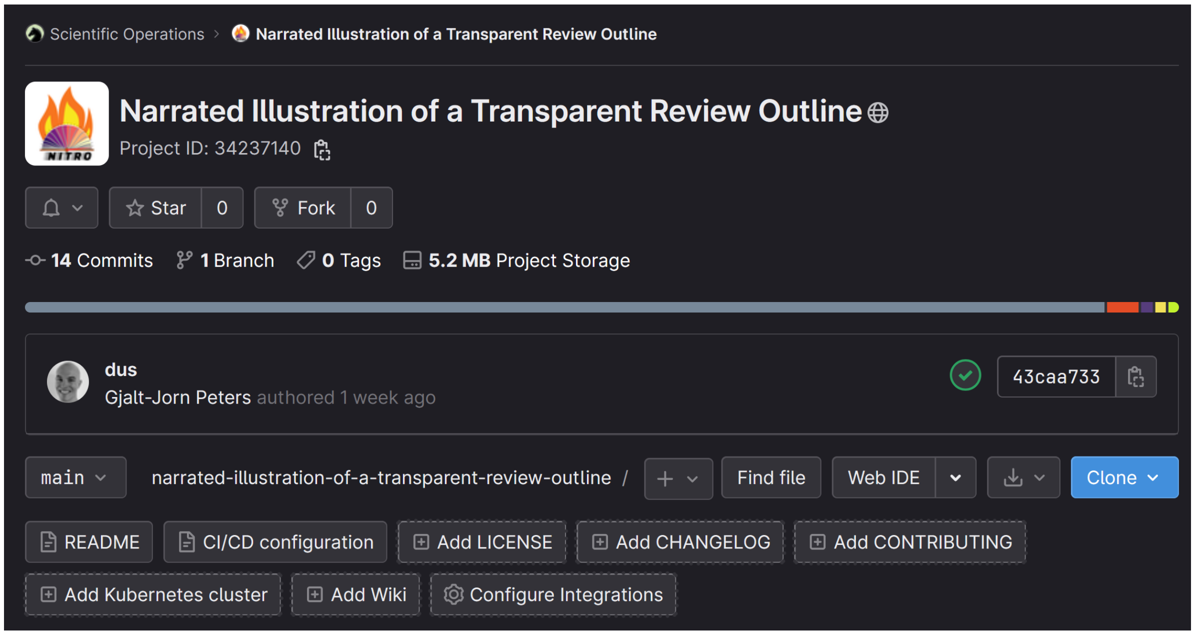
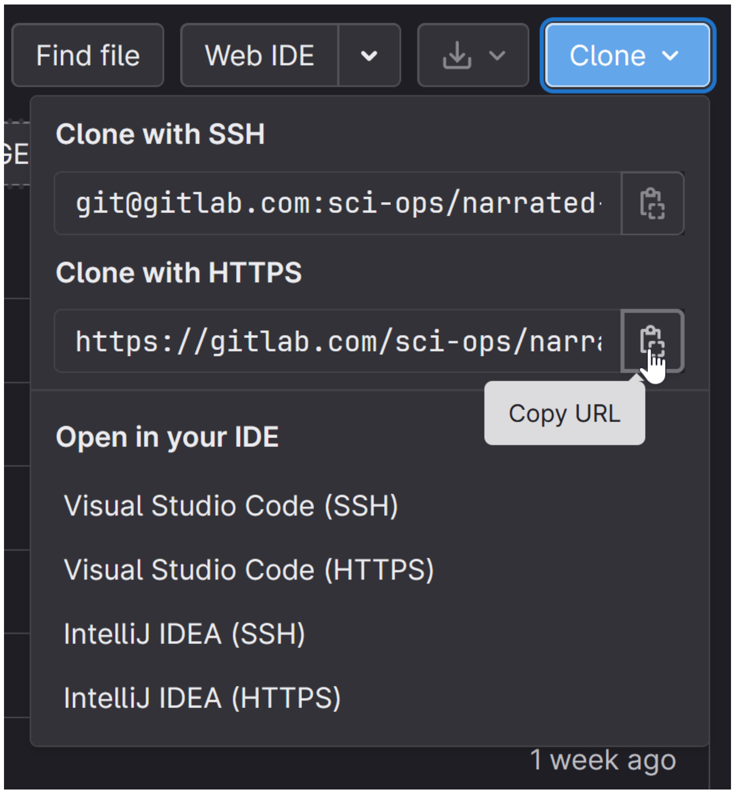


Figure 16.1: An overview of the NITRO repository in GitLab

In your browser, navigate to the URL of the Git repository (e.g., the URL of the NITRO repository). There, click the "Clone" button.

In RStudio, klik links bovenin op "File" en dan "New Project"

Kies "Version Control" en dan "Git"



```
git@gitlab.com:sci-ops/narrated
```

```
https://gitlab.com/sci-ops/narr
```

Figure 16.2: Cloning the NITRO repository in GitLab

Copy-paste de URL

Druk op Tab; hiermee gaat de cursor naar het volgende veld waar de directorynaam wordt gespecificeerd. Als het goed is “autovult” hij die met de naam van het repository (in bovenstaand voorbeeld, “verlicht-scoping-review-1”)

Kies in het derde veld een plaats om die directory aan te maken en het project naartoe te ‘clonen’. Neem bij voorkeur een lokale directory, dus niet een subdirectory van een directory van een cloud-dienst (e.g. OneDrive, DropBox, etc). Bij de OU zijn de Documenten directory en de Desktop wel subdirectories van een synchronisatiedienst (met de OU server), dus als je op een OU laptop zit, kies dan een directory op de D-drive. R opent nu je project.

16.2 For every source

1. Open the source you want to extract (e.g. a PDF of an article)
2. Look up the unique source identifier for this source. If the source has a DOI, then go to https://shortdoi.org to produce the corresponding ShortDOI. If it doesn’t have a DOI, the source identifier is the QURID, which you can get from the bibliographic database you used for the screening.
3. Open RStudio or whichever text editor you use
4. Navigate to the directory holding the Rxs template (probably “`extraction-Rxs-spec`”)
5. Open this file.
6. Save it again with a new name, specifically a name constructed as follows:

    1. Family name first author (with all characters other than Latin leters (a-z) removed)
    2. An underscore (`_`)
    3. sourceId (shortdoi or QURID)
    4. An underscore (`_`)
    5. Year of publication of the source
    6. An underscore (`_`)
    7. Extractor identifier (your identifier)
    8. The extension (`.rxs.Rmd`)

7. An example filename is `batty_2020_j58m_fm2.rxs.Rmd`
8. Then, start completing the Rxs file: scroll through it from top to bottom, and insert the value of every entity that you extract from the PDF
9. If you use Git, then once you’re done, you can push your Rxs file with:
   `git pull; git add . ; git commit -m "Commit boodschap" ; git push`

16.3 Coding an extracted entity

When planning the extraction, you decided for each entity where on the coding-categorization continuum you would extract it (see Chapter 6). The entities that you decided to extract literally from the sources have to be coded.

This coding consists of attaching code identifiers to the entity values for a given entity. If you code entities, the extracted values will almost always be literal text strings that were integrally copied from the sources.

Examples of entities that you may choose to extract integrally and then code are the definitions authors use for a concept; or their description of their sampling procedure; or their reasoning underlying a certain decision; or the way they phrase the research questions or their conclusions.

During the coding, you look for patterns in these extracted texts, and you then attach codes to make those patterns machine-readable (which will make it possible to combine this information with the other information you extracted).

16.3.1 Your codebook

As you attach more codes, you develop your codebook. A codebook is a document that defines each code. Typically, for each code, you document the following:

- **A code identifier**: Like entity identifiers, these can only consist of lower and upper case Latin letters (`a-z` and `A-Z`), Arabic digits (`0-9`), and underscores (`_`), and must always start with a letter. Valid identifiers are `socialNorms`, `broadConclusion`, `conditionalStatement`, `sampling_purposive`, and `rhetoric_authorityArgument`.
- **A code label**: This is a very short human-readable title for the code. Unlike identifiers, labels are pretty much unconstrained, so valid labels are `Social Norms`, `Drawing of a Broad Conclusion`, `Statement conditional upon other statement`, `Sampling strategy: purposive`, and `Rhetoric: use of authority argument`.
- **A code description**: This is a longer description of the concept captures by the code. This can be quite long, multiple paragraphs even, and will typically become more comprehensive as you code more data and so develop your codes further and further. The code description determines when a code applies to a given bit of data, together with the coding instruction.
- **A coding instruction**: This is an explicit instruction as to when this code should be attached to a data fragment. Unlike the code description, which describes the concept that the code captures (e.g. a psychological construct, or a procedural element of a study, or a reasoning strategy), the instruction is very operational. It described what a coder should look for in the data to decide whether this code should be applied. It typically also describes edge cases and uses those to refer to other codes, for example: "Use this

code for expressions that relate directly to authors' expectations regarding the outcome variables of the study. However, do not use this code for expectations regarding variables that are included in the design as predictors or covariates. In those cases, instead attach the codes with identifiers `expect_varPredictor` and `expect_varCovariate`."

In addition to these four basic characteristics, it is often beneficial to include examples of data that should or should not be coded with the code. Specifically, it is helpful to include four types of examples:

- examples of data fragments that should be coded with the code according to the code description and coding instructions (matches);
- examples of data fragments that should *not* be coded with the code according to the code description and coding instructions (mismatches);
- examples of data fragments that are relatively ambiguous to classify (edge cases);
- examples of data fragments that are very easy to classify (core cases);

Developing these four types (core case matches, core case mismatches, edge case matches, and edge case mismatches) help establish what you consider to be clear core examples of what should and should not be coded, as well as where the codes' conceptual boundaries lie.

An example of a codebook in spreadsheet format is available here. Note that this example is for a codebook as applied to qualitative data as produced in individual interviews with participants, so the types of codes will probably be quite different from the types of codes you use in systematic reviews.

When developing your codebook, you aim to describe the codes so explicitly, accurately, and comprehensively that others can also use the codebook. Ideally, everybody who uses the codebook codes the data in the same way. If there are resources for this in your project, it pays to have multiple coders and develop the codebook together. If you manage to create a codebook that indeed can be applied by others to arrive at the same codings (i.e. the same codes applied to the same data fragments), you know you have described the codes sufficiently clearly to be transparent about the relevant parts of the data as captures by the codes.

16.3.2 Inductive versus deductive coding

Coding occurs on a spectrum from fully inductive to fully deductive, with most cases being somewhere in the middle.

Fully inductive coding means you have no preliminary ideas about what kind of codes you may encounter. This is also called 'open coding'. In this case you start without a codebook, and you create the codebook as you develop your codes from scratch.

Fully deductive coding means you start out with a full codebook. If you really only code deductively, that implies that the codebook will not be updated, because such updates suggest that you still learn about the concepts you're coding, which would suggest you work partly inductively after all. In the context of a systematic review, it's unlikely (but not impossible) that you decide to code an entity deductively. After all, if you already know pretty much everything about it, you can often capture this knowledge in categorical entities, and so you categorize upon entity extraction (and do not extract raw source data such as text strings).

Often, you have some idea as to which codes you'll probably use, but you're also often open to changing these (maybe even completely if necessary), so it's quite common to have a very rudimentary codebook when you start coding, but to heavily develop this further during coding.

16.3.3 Coding in the ROCK standard

The `{metabefor}` package can export entities to and from the ROCK format (see this manual page). The ROCK format is the format for the Reproducible Open Coding Kit: a standard to store coded qualitative data in a format that is simultaneously human- and machine-readable (for more information, see http://rock.science).

When importing ROCK data that was previously exported by the `{metabefor}`, the codes can be combined with the other extracted data. This realizes a fully transparent process, where it is clear which raw data were coded, which codes were applied to which entities (using which codebook), and how these are combined with the rest of the extracted data.

The ROCK standard consists of plain-text files (just like `.Rxs.Rmd` files are plain text files). These can be opened with any software that can edit text files, such as RStudio, Notepad++, or stock applications that come with your operating system, such as Notepad on Windows and TextEdit on MacOS. To code a data fragment with a code, simply add the code identifier for that code at the end of the same line as the data fragment, in between two square brackets.

This is an example of a coded data fragment:

```
This is some text in the fragment. [[this_is_a_code_identifier]]
```

It is also possible to attach multiple codes:

```
This is some other text fragment. [[code1]] [[code2]]
```

This is an example of some exported and then coded entity values:

```
[[rxsSourceId=gpg36g]]
[[rxsEntityId=population]]

hotel employees on the Canary Islands [[employees]]
```

```
[[rxsSourceId=qurid_7pdb7n68]]
[[rxsEntityId=population]]

undergraduates from a public university [[students]]


[[rxsSourceId=gnt2hz]]
[[rxsEntityId=population]]

healthcare professionals working in a in a hospital in the Northern Italy [[employees]]


[[rxsSourceId=knfh]]
[[rxsEntityId=population]]

employees of an Italian subsidiary of a European multinational company  [[employees]]
        active in the food producer sector
```

The two codes at the start of every fragment are added by the `{metabefor}` package when it exports the entity values to the `.rock` file. The `rxsSourceId` is the source identifier of the source from which the following entity value was extracted. The `rxsEntityId` is the identifier of the entity from that that entity value was extracted (in this case, we're coding the authors' description of the population they studied).

The codes that were added by the researcher have code identifiers `students` and `employees`, and the fact that these are not just words (parts of the data) but codes that have been attached to the data is signified by the double square brackets surrounding each code identifier.

During coding, the things that costs most time is thinking. Thinking about whether a given code applies; thinking about whether the code book should be updated; and if so, how. The time you spend actually attaching the code is often negligible compared to the time you need for thinking. Coding is mostly an intellectual effort, and the operational side (i.e. adding the code identifier to the `.rock` file) is relatively small. Therefore, often you might as well type the code identifier in a text editor. However, if you want, there is a rudimentary interface to add codes called iROCK. You can access it at https://i.rock.science. If you use iROCK, make sure you download the coded file again and store it locally: if you forget to download it and close your browser, you lose all your work. A part of the workshop about the ROCK explains how to work with iROCK, in case you want more information: https://rock.science/workshop/2hr.

16.3.4 After the coding

Once you finished coding a source, it can be imported again and the codes will be added to the rest of the extracted data. You can then include this in datasets you want to extract from the object with extracted data. If you are doing this as a student, your supervisor will most likely handle the importing and the production of the dataset, which they will then send to you in `.xlsx`, `.csv`, `.obv`, or `.sav` format.

17 Ops for the synthesis

Part IV

References and Appendices

18 Example Projects

18.1 NITRO

The best example project to check out might be the Narrated Illustration of a Transparent Review Outline, or NITRO. NITRO is a bare-bones but narrated illustration of an (excessively simple) scoping review.

It is hosted in this Git repository at GitLab. That means that you can easily download all files (for example in this zip archive) or clone the project. The rendered R Markdown file is served here by GitLab Pages.

18.2 Ongoing projects

These are the Rxs specification spreadsheets and Git repositories for a number of ongoing projects.

- VERLICHT Scoping Review 1 - Financial Literacy: Rxs Spec | Git repo
- VERLICHT Scoping Review 2 - Green Spaces: Rxs Spec | Git repo

18.3 Preregistrations

Note that you have to open the PDF in these preregistrations!

- Extending the Earcheck: https://osf.io/v5jb8
- VERLICHT Scoping Review 1 - Financial Literacy: https://osf.io/nu69h
- A Scoping Review of Social Support and Academic Achievement - https://osf.io/azrkj | Update: https://osf.io/mc8ny
- Financial literacy, constituent behaviors and associations with financial resilience - a scoping review: https://osf.io/2c6rz
- A Systematic Review of Works in Quantitative Ethnography https://osf.io/by7me

18.4 Finished real-life projects

In addition, this is a list of example `metabefor` projects. They vary in scope: for example, some are bachelor's or master's thesis projects, others are done in the context of a PhD. thesis or are community-run living reviews.

- Drugs and Crime Scoping Review, GitLab
- Habituation versus Inhibition in Exposure Therapy, OSF
- A Systematic Review of Works in Quantitative Ethnography, OSF
- Extending the Earcheck Living Review, OSF, GitLab, GitLab Pages, Rxs Spec

19 Glossary

This glossary contains definitions of terms used in when describing evidence synthesis (e.g., systematic reviews) as well as terms introduced by `{metabefor}`.

Block core The block core is is a part of an Rxs file: it is the part of an *entity block* where the extracted value for an entity is specified during extraction. This is explained in detail in the Execution: Extraction chapter (Chapter 12 in this version of the book).

Block end marker The block end marker is a part of an Rxs file: it is a sequence of characters that denote that en *entity block* ends (which was started by a *block start marker*). This is explained in detail in the Execution: Extraction chapter (Chapter 12 in this version of the book).

Block start marker The block start marker is a part of an Rxs file: it is a sequence of characters that denote that en *entity block* starts (which is then ended by a *block end marker*). This is explained in detail in the Execution: Extraction chapter (Chapter 12 in this version of the book).

Container entity An *entity* that does not store a value (i.e. cannot itself be extracted) but serves to organise other entities in a hierarchy. For example, a container entity could be “Provenance”, containing “regular” entities “Authors”, “Region”, and “Date”. For the latter three, values are extracted, and though those three together can form a source’s provenance, the entity provenance itself has no value: it just contains the other three entities.

Cluster A set of entities specified in a list in a parent entity: see *clustering entity* and *clustered entity*.

Clustered entity An *entity* that is specified in a list within a *clustering entity*. To specify that an entity is a value list entity, specify as its parent entity a *clustering entity*. Clustering entities are presented in a list in the `Rxs template` (within the parent entity). The coding guides that are normally presented above and below every entity are then concatenated and shows after it. This has two drawbacks. First, if word wrapping is active, it makes the Rxs file look quite messy and potentially intimidating. Second, since the descriptions and examples are less salient, this is a bit less straightforward for extractors, so may require more intensive training.

Clustering entity An *entity* that as its value contains a list of *clustered entities*. To specify that an entity is a clustering entity, set the `list` column in the *Rxs specification* spreadsheet to `TRUE`. All entities that specify that clustering entity as parent will then be presented in a list within that entity in the `Rxs template`. They are useful to efficiently extract closely related information. For example, characteristics of a variable that is reported on in a source; various possible effect size measures that could be reported to describe an association; or closely related characteristics of a procedure that is described in a source.

Entity Something that can be extracted from a *source*: examples are effect sizes, sample sizes, study design characteristics, author names, publication dates, journal names, whether sampling strategies were justified, or the literal text used to phrase the main conclusion. Once an *extraction script* has been parsed, entities are also commonly referred to as *fields* that hold *values*. However, note that entities can be repeating: they refer to a type of thing that can be extracted, not to any specific thing that has been extracted, whereas a field always holds a specific extracted value. For example, it is possible to specify a *clustering entity* called `result_association` used to extract various effect sizes or other statistics that may be specified to describe an association between two variables, each as a *clustered entity*, e.g. entities called `pearson_r` and `cohen_d`. In any given extraction script, multiple instances of that `result_association` entity can be extracted, each being represented by a different `entity node`, and each of those *clustering entities* holding multiple *clustered entities* or *fields*. Each entity node holds a `pearson_r` field and a `cohen_d` field, but in the *extraction script specification*, only one entity is specified for each of `result_association`, `pearson_r`, and `cohen_d`. To refer explicitly to the entity as a "thing that can be extracted", you can use the terms *entity specification* or *extractable entity*. To refer explicitly to an entity that has been extracted, use `field` or `extracted entity`.

Entity block An entity block is a part of an Rxs file: it is a block in that plain text file with the information about an entity to extract. It starts with a *block start marker* and ends with a *block end marker*. This is explained in detail in the Execution: Extraction chapter (Chapter 12 in this version of the book).

Entity container See *container entity*.

Entity description sub-block An entity description sub-block is a part of an Rxs file: it specified the label and the description of the entity to extract in the *entity block* it is a part of. This is explained in detail in the Execution: Extraction chapter (Chapter 12 in this version of the book).

Entity identifier The unique identifier of an *entity* as specified in the `identifier` column of the *extraction script specification*.

Entity node In the `data.tree` object representing each extracted source, *entities* are stored as nodes in a hierarchical tree. In this tree, each entity correponds to one or more entity nodes, except for *clustered entities*, which are stored as *values* in a list in the entity node representing their *clustering entity* (i.e. their parent entity).

Entity node identifier The unique identifier of an *extracted entity* (i.e. an instance of an *entity specification*)

Entity specification A way to explicitly refer to an entity in its generic definition, as a class of things that can be extracted, as opposed to as a *field* or *entity node*, which represents an extracted entity. This distinction is important because entities can be repeated (see *entity*).

Extractable entity See *entity specification*.

Extracted entity A way to explicitly refer to an entity that has been extracted, as an instance of the *entity specification* or *extractable entity* (a generic description of a class of things that can be extracted). Extracted entities are *entity nodes* unless they are *clustered entities*, in which case they are stored as *fields* holding *values* in a *list* in their parent entity (i.e. their *clustering entity*). Extracted entities that are entity nodes are either *container entities* (if they don't store a value themselves, but instead are used to organise other entities), *fields* (if they hold a single *value*), or *clustering entities* (if they hold a *list* specifying the values for multiple *clustered entities*).

Extraction The act of registering an *entity* in an *extraction script* (or extraction form, if `metabefor` isn't used) in the form of a text string or an entity conform the coding instructions (as specified in the *Rxs specification* if `metabefor` is used).

Extraction script (`.Rxs` file) An R Markdown file with the extension `.Rxs.Rmd` that is a completed *Rxs template*, and as such, set up such that it can be parsed by `metabefor`.

Extraction script specification A spreadsheet specifying which entities to extract, how they are hierarchically structured, which *value templates* they use, and their extraction instructions (or coding instructions).

Field A synonym of entity that is used when it is referred to as a specific value holder in the context of a completed extraction script. For example, when in an *extraction script specification* an entity is specified to extract the source's publication date with entity identifier `pub_date`, in the parsed *extraction script*, the value of that date can be referred to as being stored in the `pub_date` field. A field represents an instance of an *entity*: see that definition for more details.

List A list of multiple *clustered entities*. Lists are an efficient way to extract closely related entities, such as measurement details about a given variable, descriptives for an outcome, or various effect size metrics that may be reported.

Rxs file A file with an R extraction script. One example is an *Rxs template* as produced by [metabefor::rxs_fromSpecifications()], but usually these Rxs files will be completed versions of that template, each containing one or more extracted *entities* from a *source*. Rxs files have the extension `.Rxs.Rmd`.

Rxs specification See *extraction script specification*.

Rxs template The structured R Markdown file produced by `metabefor` after it parsed the *Rxs specification* spreadsheet.

Source Something that *entities* can be *extracted* from, such as an article, book, case law, report, webpage, or other source.

Value A value; usually a single number or character string, but values can be more complex, too, e.g. vectors of numbers of strings, or even tables (matrices or arrays).

Value description sub-block A value description sub-block is a part of an Rxs file: it specified the value to extract for the entity in the *entity block* it is a part of. This is explained in detail in the Execution: Extraction chapter (chapter @ref(execution-extraction) in this version of the book).

Value list A list of *values*; a synonym for all *extracted clustered entities* stored in an *extracted clustering entity*.

Value templates In an *Rxs specification*, a template for a data type that can be extracted, specifying the values that are allowed, the default value, instructions, and validation rules.

20 Notes

21 Drafts

roles researches