

The OSG Lecture Bundle

User Manual

Matthias Werner

October 23, 2026

Contents

1	Introduction	3
1.1	Idea	3
1.2	So what exactly <i>is</i> OSG Lecture?	4
1.3	Polymorphism in OSG Lecture	5
2	Short Tutorial	8
3	Concepts	19
3.1	Project	19
3.2	Course Unit	19
3.3	Directory Structure	20
3.4	Modi	21
3.5	Document Types and Profiles	23
4	To Shape Content	25
4.1	Titles and Metadata	25
4.2	Presentations	26
4.3	Remarks and Source Credits	26
4.4	Multilingual Documents	27
4.5	Selecting Content with OSG Lecture Modes	28
4.6	Lists as Continuous Text	29
4.7	Two-Column Content	30
4.8	Cross References	30
4.9	Literature	31
4.10	Command Reference	32
5	Series	36
5.1	Continuing Counters	36
5.2	Integration	37

Contents

5.3	Wrapper Document	37
5.4	Tags, Links, and Fallback	38
5.5	Reference Summary	38
6	Standalone	40
7	Project Configuration	42
7.1	The Project Manifest	42
7.2	BundlePreset	43
7.3	User Defaults	44
7.4	L ^A T _E XConfiguration	45
7.5	Document Profiles	46
7.6	Extending the Mode Graph	47
7.7	Configuration Reference	48
8	OLLM	51
8.1	Building: From One Instance to a Series	51
8.2	Inspecting and Diagnosing	52
8.3	Cleaning and Maintaining State	52
8.4	Deployment	53
8.5	Creating a Project and General Options	54
8.6	Complete Command Summary	54
	Index	57

1 Introduction

1.1 Idea

Teaching materials come in many forms and versions, such as lecture notes, slides, or handouts, and may be available in different languages. Yet the content is often the same or similar, and the structures—for example, the order of the teaching and learning units—should be consistent. In other words: The same thing (here semantic information) should appear in different forms. In computer science or chemistry, this concept is also known as *polymorphism*¹

At the same time, teaching materials are generally revised on an ongoing basis. Individual teaching units are updated, reorganized, or removed entirely, while new ones are added. Maintaining the desired consistency throughout this process is difficult. The *OSG Lecture* Bundle aims to support instructors in this effort. It is based on a central idea:²

Maxim

Redundancies should be kept to a **minimum**.

Where redundancies are necessary (for example, with different versions of a text), it is easier to maintain consistency if related source material is located as **close together** as possible.

Therefore, *OSG Lecture* is intended, amongst other things, to enable:

- Generating different types of documents from the same source;
- easy reorganization of teaching units;
- different language versions from the same source, wherein shared data does not need to be presented redundantly;
- support for creating and revising individual instructional units that can be easily integrated (e.g., lecture notes chapters into a comprehensive set of lecture notes);
- rapid deployment of the various types of document.

This was based on the procedures and workflows that are currently standard practice at the Operating Systems Group at Chemnitz University of Technology. If you do not work there and, for example, have different procedures, *OSG Lecture* might still be useful to you: individual packages within the bundle can also be used independently of the others.

1. Perhaps *poly-representation* or even *multimodality* would be even more appropriate terms, but the former isn't very well established, and with the latter, I'd probably have to discuss with cultural studies scholars and semioticians whether different languages should be understood as *modes*. Besides, I'm a computer scientist, so polymorphism seems like a natural choice.

2. I am well aware that not everyone shares this guiding principle; a TeXgiant like Joseph WRIGHT, for example, is skeptical of this idea (<https://github.com/josephwright/ltx-talk/discussions/68#discussioncomment-13755453>), and he can cite good reasons for his skepticism. So if you believe that such reasons outweigh the benefits, do not use *OSG Lecture*.

OSG Lecture is the successor project to **OSG Beamer**,³ which has evolved from a theme and a handful of supplementary macros for the widely used Beamer class.⁴

1.2 So what exactly *is* **OSG Lecture**?

OSG Lecture is a L^AT_EX bundle (more specifically, a LuaL^AT_EX bundle) that brings together various ideas and approaches for creating teaching materials. None of these ideas or approaches is particularly new.

- **osglecture** is a metaclass that delegates most of the work to other classes, much like **beamerswitch**,⁵ for example. In addition, **osglecture** itself provides several features, including a cross-document reference system and even some formatting environments.
- **langselect** is a package for selecting which text is to be included in the target PDF depending on the language, similar to **comment**⁶ or **multilang**.⁷ The language actually used can also be specified externally.
- **osglecture-modes** generalizes the concept of presentation modes familiar from **beamer**⁸ or **ltx-talk**.⁹ It therefore does something similar to **langselect**, but at the structural rather than the language level. This makes it possible to use modes outside presentations, too, with the familiar `<>` syntax of **beamer** or **ltx-talk**.
- **tagpax** imports complete pages from tagged PDF documents into a PDF document that is itself tagged. Besides the tag structure, links, destinations, and annotations are retained. In the context of **OSG Lecture**, it makes it possible to compile, for example, individual lecture notes chapters independently and combine them into a complete set of lecture notes without having to perform one “large” compilation run.¹⁰
- **OLLM**: yet another L^AT_EX build tool that is tailored specifically to **OSG Lecture**. It works closely with **osglecture** and the other packages mentioned above. Internally, **OLLM** invokes **latexmk** (it originated as an rc file for **latexmk**) which performs most of the work.

While the packages mentioned above form the core of the bundle, several additional packages have proved useful for creating teaching materials. They do interact with the core packages, but contribute more directly to the design of the documents than to implementing the polymorphism approach, although they make use of it.

3. <https://github.com/tuc-osg/osgbeamer>

4. <https://ctan.org/pkg/beamer>

5. on CTAN as beamerswitch: <https://ctan.org/tex-archive/macros/latex/contrib/beamerswitch>

6. on CTAN as comment: <https://ctan.org/tex-archive/macros/latex/contrib/comment>

7. on CTAN as multilang: <https://ctan.org/tex-archive/macros/latex/contrib/multilang>

8. on CTAN as beamer: <https://ctan.org/tex-archive/macros/latex/contrib/beamer>

9. on CTAN as ltx-talk: <https://ctan.org/tex-archive/macros/latex/contrib/ltx-talk>

10. For my lecture notes, such a large run can sometimes take more than an hour.

- `osgstyler` seeks to reinforce the core L^AT_EX principle of separating content from presentation through a central semantic configuration. This makes elements such as colors, fonts, spacing, highlighting, and blocks easier to design consistently as well as easier to replace. It works well in conjunction with `osglecture-modes`.
- `ltxtalk-theme` provides a common framework for themes of the `ltx-talk` class. It uses `osgstyler` for design within slides.
- `lttheme-tuc-2019` is the unofficial implementation of my university’s official presentation layout for `ltx-talk`. It uses `ltxtalk-theme`.
- `semcat` marks content with freely configurable semantic categories such as “background”, “technical details”, or “review”. Categories may appear inline, as separate paragraphs, or as highlighted blocks and may be indicated by colours, symbols, or QR codes.
- `ansiterm` provides two semiverbatim environments for displaying computer terminal windows that can interpret ANSI escape sequences.¹¹ Terminal transcripts may be supplied directly or produced in a shell at compilation time and cached for later runs. Their text remains accessible in tagged documents.
- `osglistings` typesets source code using `minted`.¹² It supports several presentations, external sources, and semantic marks and explanations within code; its output can also be used in tagged documents.
- `tagbridge` supplies tagging support for third-party packages while they do not yet provide sufficient PDF/UA support themselves. This currently concerns QR codes and TikZ graphics in particular.
- `tagdirtree` typesets directory trees as genuine nested lists rather than as graphics. Files, directories, and archives thus remain readable text, and their hierarchy is available to assistive technology.

1.3 Polymorphism in OSG Lecture

It is a long-standing T_EX concept to render text differently depending on the context. The T_EX variant ConT_EXt, for example, derives its name from this concept. `OSG Lecture` attempts to expand the application of this concept a bit further. In this regard, `osglecture` is a *metaclass*: Unlike most L^AT_EX classes, it does not specify how the final typeset documents should look,¹³ but rather provides a common interface and then delegates the work to classes such as `article`, `scrarticle`, `beamer`, or `ltx-talk`.

The following is a mini-showcase designed to demonstrate how different instances are created from a single source. Its sole purpose is to give the user a brief glimpse of the possibilities of

11. See, for example, https://en.wikipedia.org/wiki/ANSI_escape_code.

12. on CTAN as `minted`: <https://ctan.org/tex-archive/macros/latex/contrib/minted>

13. However, there are packages in the `OSG Lecture` bundle that do just that

context-dependent polymorphism in [OSG Lecture](#)—the discussion of how it works and the mechanisms used will follow later.

```

1 \begin{frame}
2 \frametitle{\lende{One Source, Many Presentations}{Eine Quelle, viele Darstellungen}}
3 \begin{presitemize}
4 \item \lende{a unit contains the subject-matter content}%
5     {eine Lehreinheit enthält den fachlichen Inhalt}
6 %
7 \item \lende*{target @1 produces a presentation}%
8     {das Target @1 erzeugt eine Präsentation}{\emph{talk}}%
9 %
10 \anditem \lende*{target @1 produces a long form}%
11     {das Target @1 erzeugt eine Langform}{\emph{script}}%
12 %
13 \thereforeitem \lende{we \finite{need} to maintain changes only once}%
14     {wir \finite{müssen} Änderungen nur an einer Stelle pflegen}%
15 %
16 \item \lende{language-independent data needs to be entered at one place}%
17     {sprachunabhängige Daten müssen nur einmal angegeben werden}%
18 %
19 \only<article>{\lende{since they can be inserted into the text using @-parameters}%
20     {da sie über @-Parameter in den Text eingefügt werden können}.%
21     }%
22 \end{presitemize}%
23 \end{frame}
24 \lende*{Text written outside the @1 environment is reserved exclusively for the long form.}%
25 {Außerhalb der @1-Umgebung geschriebener Text ist exklusiv der Langform vorbehalten.}{\texttt{frame}}

```

Figure 1: Common source code

1 Introduction

One Source, Many Presentations

- A unit contains the subject-matter content
- Target *talk* produces a presentation
- Target *script* produces a long form
- We need to maintain changes only once
- Language-independent data needs to be entered at one place

Eine Quelle, viele Darstellungen

- Eine Lehreinheit enthält den fachlichen Inhalt
- Das Target *talk* erzeugt eine Präsentation
- Das Target *script* erzeugt eine Langform
- Wir müssen Änderungen nur an einer Stelle pflegen
- Sprachunabhängige Daten müssen nur einmal angegeben werden

(a) Slides in English

(b) Slides in German

1

Eine Lehreinheit enthält den fachlichen Inhalt. Das Target *talk* erzeugt eine Präsentation und das Target *script* erzeugt eine Langform, weshalb wir Änderungen nur an einer Stelle pflegen müssen. Sprachunabhängige Daten müssen nur einmal angegeben werden, da sie über @-Parameter in den Text eingefügt werden können. Außerhalb der `frame`-Umgebung geschriebener Text ist exklusiv der Langform vorbehalten.

(c) Version for script-Target

Figure 2: All variants were created from the source shown in Fig. 1: For the slides in different languages (a) and (b), the structure, sequence of statements, and emphasis remain the same in each case; the output for the script (c) is equivalent in content but more detailed and in sentence form.

It is well known in L^AT_EX that text markup can vary and thus be polymorphic. However, **OSG Lecture** provides convenient support for this through specific profiles.

- 1 Ein `\emph{Seitenfehler}` unterbricht den normalen Kontrollfluss.
- 2 Das Betriebssystem muss anschließend `\textbf{den fehlenden Seitenrahmen}`
- 3 `bereitstellen`}.

Figure 3: Ordinary author markup can be polymorphic as well

Ein *Seitenfehler* unterbricht den normalen Kontrollfluss. Das Betriebssystem muss anschließend **den fehlenden Seitenrahmen bereitstellen**.

Ein *Seitenfehler* unterbricht den normalen Kontrollfluss. Das Betriebssystem muss anschließend **den fehlenden Seitenrahmen bereitstellen**.

(a) Screen profile

(b) Print profile

Figure 4: Realization of the code from Figure 3: The same semantic markup with different profiles.

The following short tutorial demonstrates why this kind of polymorphism is needed in the first place.¹⁴

¹⁴ The idea of presenting a tutorial as a story was inspired by Till TANTAU and his tutorials on [Beamer](#) and [TikZ](#).

2 Short Tutorial

This tutorial follows Nora. Next semester, Nora will be teaching the lecture “Operating Systems” for the first time. The first lesson will cover storage. Nora wants to dive right into storage virtualization. That might be a bit optimistic, but let’s not let that stop us here.

Although the degree program in which the course is offered is German-language, Nora knows that many visiting students will be attending who are more comfortable with English than with German. That’s why she wants the slides to be in both languages. She has also promised to provide a short set of lecture notes. For the lecture itself she also wants a presenter’s version showing the current and next slide together with her own remarks.

Several document instances for what is initially just a single hour of instruction: Nora already suspects that having separate source files wouldn’t be a particularly good idea.

The tutorial does not describe every option in the bundle. Instead, it shows the things Nora actually needs for her first small course.¹⁵ At the end, one course unit will produce slides, a presenter’s version, and lecture notes, each in two languages.

Nora’s Problem. Nora starts off the way one would probably intuitively begin without [OSG Lecture](#). She creates `slides-de.tex`, copies the file to `slides-en.tex`, and begins the translation. For the script, she later adds `script-de.tex`. That works. However, shortly afterward, she realizes that she forgot something important in the description of the difference between *paging* and *segmentation*. But while she’s editing, her phone rings, and after the call, Nora has to figure out which of the files she made the correction in.



Figure 5: Copies are easy to create and awkward to maintain.

Nora therefore decides to use [OSG Lecture](#) and to maintain only one subject-specific source. The languages should be closely aligned within it; slides and lecture notes should also be derived from this source. This is exactly what will be set up below.

A Home for the Course. First, the course needs a directory. Nora could put every file directly into it. Instead, she immediately chooses two subdirectories: one for shared settings and one for the first course unit. Giving the unit the number 010 rather than 001 initially seems unnecessarily generous. Later, however, she can insert a new unit named 005-Overview before it without renaming everything.

¹⁵. If you’d rather get right into the details, you can continue reading directly in section 3.

```
mkdir -p Noras-os/Include Noras-os/010-Memory
cd Noras-os
touch 010-Memory/main.tex
```

The result is not particularly impressive yet:



At least it is now clear what belongs to the whole course and what belongs only to the “Virtual Memory” unit.

Nora Orders Some Outputs Nora now has directories, but no way to turn her single source into the documents she wants. This is the job of **OLLM**, the *OSG LaTeX Lecture Maker*. **OLLM** is the bundle’s build tool: it selects the document type and language, starts the appropriate $\text{\LaTeX}$ run, and manages the resulting files. Nora will therefore not tell **OLLM** the subject matter of her lecture, but she will tell it which outputs she expects.

A specific selection of a teaching unit, target, and language is referred to below as *instantiation*. Nora is already familiar with this term from object-oriented programming. But unlike in that context, the result of an instantiation is not an object but a *document instance*. For example, the combination “Lesson memory, Target slides, Language en” generates the English slide instance of this lesson. This precisely specifies what **OLLM** is to generate later.

Nora would prefer to start **OLLM** immediately by typing `ollm slides`. However, the tool cannot yet know which languages the course offers or whether Nora wants lecture notes at all. This information goes into a manifest named `ollmconfig.toml` in the project root. It can be read as an order form: Nora orders two language versions of the slides and two of the lecture notes.

```
schema = 2
bundle_preset = "OSG lecture/1"

[project]
id = "os-in-60"

[targets.defaults]
languages = ["de", "en"]
default_language = "de"
presentation_profile = "ltx-talk"
longform_profile = "book"
document_metadata = "enabled"
```

```
[targets.slides]
[targets.screen]
[targets.script]
```

Nora is tempted to add the title, author, and translations here as well. After all, TOML can store text. She resists the temptation: the manifest describes the build matrix, not the document's content. The title and language mapping are L^AT_EX concerns and follow in the next step.

Before continuing, Nora checks the order form:

```
ollm check
```

On her first attempt, she had written `language` instead of `languages`. OLLM rejects the unknown key. Nora prefers that to a course that's silently in German. After she corrects the typo, the manifest is valid. Even if her build environment itself is causing problems, `ollm doctor` provides a more accurate diagnosis.

What Applies to Every Unit. Nora does not want to repeat her name and the course title in every future unit. She therefore creates `Include/projectconfig.tex`. In the manifest she has already selected `ltx-talk` for presentations and the standard `book` class for the lecture notes. Both profiles use L^AT_EX's new PDF management, allowing Nora to provide her documents as Tagged PDF. She can select a suitable presentation theme independently.

The choice of classes is a project-level decision; the individual subject-specific texts within the teaching units do not need to be aware of these class names. In addition, Nora learns that documents that are not part of the presentations or handouts are called “long forms” in `OSG Lecture`. Her lecture notes are one such long form.

She writes into `/Include/projectconfig.tex`

```
\title{\ldeeen{Betriebssysteme}{Operating Systems}}
\author{Nora Example}

\LectureProjectSetup{
  languages={
    map={de=ngerman,en=british},
    load babel
  }
}
```

The profile selection belongs in the manifest rather than `projectconfig.tex`, because the base class and its need for `\DocumentMetadata` must be known before the class starts. Nora therefore also creates `Include/documentmetadata.tex` for the early PDF configuration:

```
\DocumentMetadata{
  lang = \OsgLectureRequestedLanguage,
  tagging = on
}
```

OLLM includes this file before `\documentclass` for all three targets. Visible document metadata remains in `projectconfig.tex`.

The remaining language configuration contains two different kinds of names. OLLM uses the short, portable identifiers `de` and `en`. The popular language package `babel`, by contrast, needs the more specific names `ngerman` and `british`. The `languages` key passes this mapping on to `langselect` and loads `babel`. Which languages the project offers at all is already in the manifest – OSG Lecture hands that list from there to `langselect`.

At Last, Some Subject Matter. Now Nora writes `010-Memory/main.tex`. She deliberately starts with little content. A small document that builds successfully is a more pleasant starting point than twenty pages with a missing closing brace.

```
\documentclass{osglecture}

\begin{document}
\lecture[Memory]{\ldeen{Virtueller Speicher}{Virtual Memory}}{lec:memory}

\begin{frame}
  \frametitle{\ldeen{Warum virtueller Speicher?}{Why virtual memory?}}
  \ldeen{Jeder Prozess sieht einen eigenen Adressraum.}
    {Each process sees a private address space.}
  \Note{\ldeen{Hier an die Analogie mit Zimmernummern erinnern.}
    {Recall the analogy with room numbers here.}}
\end{frame}

\begin{frame}
  \frametitle{\ldeen{Seitentabellen}{Page tables}}
  \ldeen{Eine Seitentabelle ordnet virtuelle Seiten physischen
    Seitenrahmen zu.}
    {A page table maps virtual pages to physical page frames.}
\end{frame}
\end{document}
```

The `\lecture` command begins a course unit. Its first mandatory argument is the title, and the optional argument is the short title. The second mandatory argument of `\lecture`, here `lec:memory`, is the label of the unit. It constitutes the unit’s stable identity. Nora may improve

the visible title later and may even rename the directory. She should not casually change the identity while doing so; cross-references and build state use this very value.

Nora already knows the frame environment from her earlier work with `beamer`. She often put the title in an environment argument there, but `OSG Lecture` does not permit that; the `\frametitle` command must be used instead. Nora has not seen the `\ldeen` command before. Even a quick grep search of her L^AT_EX directory finds no package defining it. Nevertheless, its purpose is intuitively clear to her: it places the two requested language variants side by side.

`\Note`, by contrast, contains a remark for Nora herself. It belongs inside the frame to which it refers. It does not appear on the ordinary slides or in the lecture notes; the presenter's version later collects it on its presenter page.

The First Build. Nora changes into the course unit and, out of habit, first tries `lualatex main.tex`. This is an understandable attempt, but it bypasses the order form: the target and language are missing. For a course series, `OLLM` is the right entry point. Nora therefore first builds exactly the output she wants to see:

```
cd 010-Memory
ollm build slides --language=de
```

The result is a German slide. It is not placed next to `main.tex`; `OLLM` keeps generated files below `.osglecture/build`, separate from the sources. Nora briefly looks in the wrong directory, but then finds the separation convincing: a damaged build directory can be removed without losing teaching material.

The English slide is now almost disappointingly easy:

```
ollm build slides --language=en
```

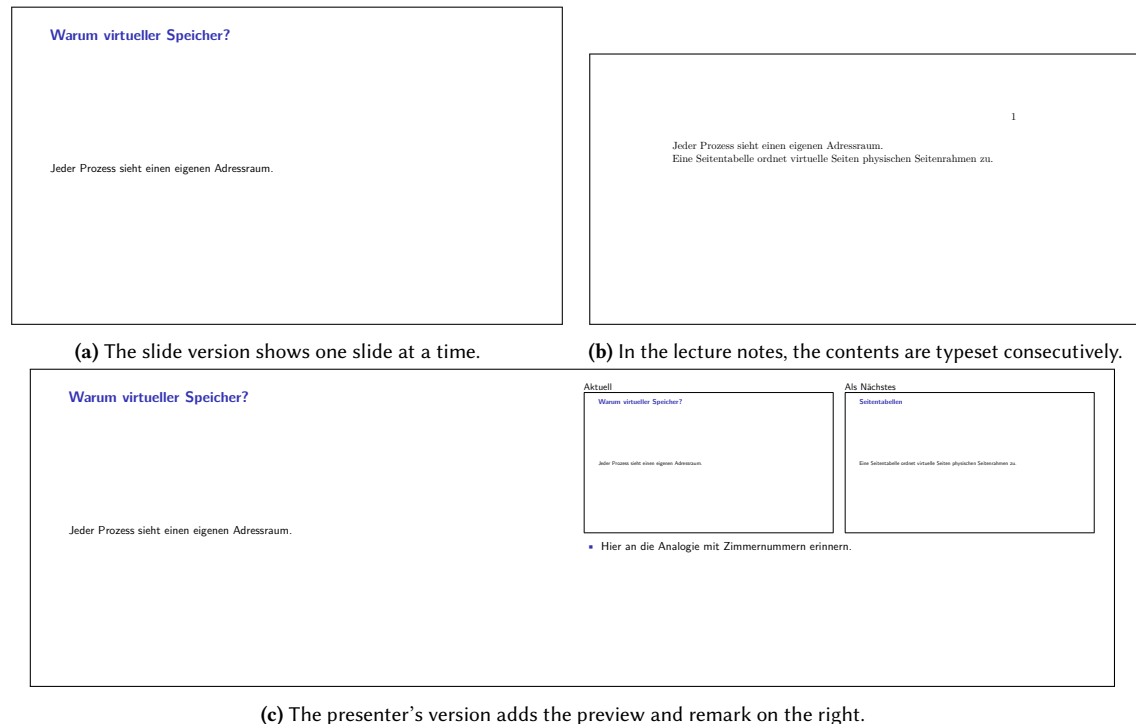
Only the selected wording changes. The structure and formula remain the same. That is precisely what Nora wanted to achieve.

A Version for Nora. For the lecture Nora also builds the screen target:

```
ollm build screen --language=de
```

This document instance uses the same slides but makes every page twice as wide. The slide for the projector is on the left; on the right Nora sees the current and next slide together with the reminder recorded using `\Note`. With an extended desktop she can distribute the two halves between the projector and her own screen. The presenter's version thus remains part of the same source without itself becoming a version for the students.

This instance is also built with tagging enabled. The additional presenter page is an aid for Nora, however, and is not part of the reading order of the slide itself.



(c) The presenter's version adds the preview and remark on the right.

Figure 6: Three document instances from the same source.

Figure 6 shows the difference between the three targets. The remark belongs exclusively in Nora's presenter version, while the lecture notes typeset both frames as continuous text.

From Bullet Points to a Sentence Nora adds two more statements. An ordinary `itemize` list would suit the slide. In the lecture notes, however, she does not want a paragraph consisting only of three bullet points. She could write two versions, but that would repeat her original problem on a smaller scale. Instead, she uses `presitemize`:

```
\begin{frame}
  \frametitle{\ldeen{Warum virtueller Speicher?}{Why virtual memory?}}
  \begin{presitemize}
    \item \ldeen{Jeder Prozess sieht einen eigenen Adressraum}
      {Each process sees a private address space}
    \anditem \ldeen{die Hardware übersetzt virtuelle Adressen}
      {the hardware translates virtual addresses}
    \thereforeitem \ldeen{Prozesse \finite{sind} voneinander isoliert}
      {processes \finite{are} isolated}
  \end{presitemize}
\end{frame}
```

On the slide these remain three entries. In the long form they become connected statements. The marked finite verb helps turn “Prozesse sind voneinander isoliert” into the German subordinate clause “weshalb Prozesse voneinander isoliert sind”. Nora finds the marking slightly unfamiliar, but still preferable to a second paragraph that might later be forgotten. And she’s glad that English is a little more regular in that regard and doesn’t mess with the verb forms so much.

A Little More Room in the Notes. The concise explanation is enough for Nora’s slide. In the lecture notes she also wants to explain where the mapping is stored. This time it is not merely the wording that differs, but the desired content. Nora therefore adds a so-called *mode section* after the frame:

```
\lecturemode<longform>{
  \deen*{Eine Seitentabelle speichert die Abbildung @1.}
  {A page table stores the mapping $v\mapsto @1.}{$v\mapsto p$}
}
```

She could have written `<script>`. `<longform>`, however, states her intention more accurately: the paragraph suits any detailed long form, not merely the script type configured today. If she later changes from `scrbook` to another long-form profile, she need not find this passage again.

Nora can write still longer explanatory passages outside the `frame` environment. She then needs no additional mode for the long form: text there is ignored in presentations.

Nora now builds the German lecture notes:

```
ollm build script --language=de
```

When comparing the results, she sees the two dimensions independently for the first time: `--language` selects the language, while the `slides`, `screen`, or `script` target selects the document form. Both change the output, but neither requires a copy of the course unit.

Did Nora Really Think of Everything? While writing, Nora usually creates only the document instance she is currently working on. This is quick, but carries a small risk: a missing brace may occur only in the English branch; a command active only in long forms may go unnoticed on the slides. Before publishing, Nora therefore builds all combinations ordered for this unit:

```
ollm build --all
```

From Build to Publication. Six successful document instances in the build directory are gratifying, but Nora's students still know nothing about them. The two presenter's versions remain with Nora; she wants to copy the slides and lecture notes to a release directory that can later be picked up by a learning platform or web server. She could copy and rename the files by hand. By the third session, she would probably have placed the English version under the German name at least once.

OLLM therefore separates *building* from *deployment*. A build creates and checks a document instance in the internal build area. A deployment copies an already successfully built instance to a publication location under an agreed name. It does not start another L^AT_EX run.

Nora adds two deployment rules at the end of `ollmconfig.toml`:

```
[deployment]
series = "units"

[deployment.types.slides]
paths = ["release/slides"]
filename = "{ordinal}-{unit}-{lang}.pdf"

[deployment.types.script]
paths = ["release/notes"]
filename = "{ordinal}-{unit}-{lang}.pdf"
```

The placeholders are filled for each document instance. For the current unit this produces, for example, `010-memory-de.pdf` and `010-memory-en.pdf`. Nora deliberately includes the language in the file name; the seemingly prettier `010-memory.pdf` would be ambiguous for two language versions.

OLLM does not create deployment directories itself. This is surprising at first, but prevents a typo in the manifest from silently creating a second publication location. Nora therefore creates the agreed directories once and then deploys all built instances of the current unit:

```
mkdir -p ../release/slides ../release/notes
ollm deploy --all
```

Nora is still in 010-Memory; the deployment therefore applies to this unit by default. **OLLM** copies only existing, successfully completed builds. If Nora has changed the text but has not rebuilt it, `deploy` is not a hidden substitute for `build`. She likes this separation: she can inspect the document instances before copying them.

At the next deployment, an older file may already exist under the same name at the destination. If it is identical, it remains untouched. If it differs, the default safety policy requires explicit approval with `--overwrite`. Nora should use this option only after confirming that she really intends to replace the published version.

One Unit Becomes a Series. For the second lesson, Nora creates the directory 020-Processes next to 010-Memory. Its `main.tex` follows the familiar pattern. What matters is not the number in the directory name, but the new logical unit ID processes:

```
\documentclass{osglecture}

\begin{document}
\lecture[Processes]{\ldeen{Prozesse}{Processes}}{processes}
\section{\ldeen{Prozessplanung}{Scheduling}}{\label{sec:scheduling}}

\begin{frame}
  \frametitle{\ldeen{Wer läuft als Nächstes?}{Who Runs Next?}}
  \ldeen{Der Scheduler wählt einen ausführbaren Prozess aus.}
  {The scheduler selects a runnable process.}
\end{frame}
\end{document}
```

The two directories are now more than accidental neighbours. The manifest and their unit IDs make them units of the same *series*. This allows `OLLN` to select their document instances together, manage their state, and resolve dependencies between them. For example, from the project root Nora builds the German script instances of the whole series:

```
ollm build script --language=de --scope=series
```

The series scope does not mean that a single large PDF is already created. Initially, two independent script instances are built. This is useful because Nora can still distribute individual chapters separately.

A Reference Across the Unit Boundary. In the chapter on memory, Nora now wants to refer to the later explanation of scheduling. An ordinary `\ref` only knows targets in the current document instance. For the cross-unit reference, Nora therefore also names the logical target unit:

```
\ldeen*{Mehr dazu in @1.}
  {More on this in @1.}
  {\olautoref[processes]{sec:scheduling}}
```

Without further settings, `\olautoref` selects the same language and document type in the target as in the source document. Thus, the German script reference leads to the German script, while the English slide reference leads to the English slides. If a reference is deliberately meant to lead, for example, from a slide to the German script, Nora can write the address as

[processes, type=script, lang=de]. She avoids page references across document boundaries: the page number of the other PDF can change independently.

The target must have been built successfully at least once so that **OLLM** knows its unit ID and reference targets. After later changes, Nora uses `--resolve` to rebuild dependent instances until they reach a consistent state:

```
ollm build script --language=de --scope=series --resolve
```

Combining the Chapters into Complete Notes. For the complete lecture notes, Nora creates a third unit. Its directory name carries the role *i* for *integration*, and it contains no further subject matter, only the wrapper document. The order of the `\includeunit` commands determines the order in the result:

```
mkdir -p 900-i-collection
```

She writes in `900-i-collection/main.tex`:

```
\documentclass{osglecture}

\begin{document}
\includeunit{memory}
\includeunit{processes}
\end{document}
```

`\includeunit` does not include the TeX sources, but the already built document instances of the same language and document type. The chapters therefore remain independently usable while the wrapper combines them into complete lecture notes. Nora first builds the chapters and then the uniquely matching integration unit:

```
ollm build script --language=de --scope=series --resolve
ollm build script --language=de --scope=collection --resolve
ollm check --scope=series
```

The final check does not build anything; among other things, it reports missing or stale reference and integration state. In a real project, Nora would of course run the same steps for the English script instance. Section 4.10 describes all variants of the reference commands; the rules for integration are in Section 5, and the complete build options are in Section 8.6.

This does not, of course, finish the small course. Nora still lacks further course units and probably more than two hours of content. But the basic pattern is in place: the manifest describes the desired outputs, the project configuration contains shared decisions, and each course unit keeps

its subject matter in one place. Nora can now add new units following the same pattern without creating new language or document copies.

Hint

If you follow the tutorial with your own files, likewise begin with one small unit and one document instance. Only once it builds should you add the second language, the long form, and finally `--all`. This keeps errors associated with the step that introduced them.

3 Concepts

After the short tutorial, the terminology and concepts of **OSG Lecture** are now discussed more systematically and in greater detail.

The name “Osglecture” has two meanings in this bundle: it denotes both the bundle as a whole and its main class. To distinguish the two, **OSG Lecture** is used below for the bundle and **osglecture** for the class.

3.1 Project

In this context, a “project” refers to a collection of related materials. In the context of our use case, this means teaching materials such as lecture slides or lecture notes, even though **OSG Lecture** can be applied to other use cases as well. In other words: An **OSG Lecture** project is a *course*.

All course materials are located in a directory that typically contains a special manifest. An exception to this rule is formed by so-called *standalone* projects, which were originally intended for one of the other use cases mentioned above, namely conference papers. This special case is discussed separately later. In what follows, however, we will focus on courses.

3.2 Course Unit

From the perspective of **OSG Lecture** a course is a series of related course units, e.g. lectures or tutorials (or both), which are typically held over the course of a semester. A teaching unit here is a didactic concept rather than a temporal one: It is entirely possible for a teaching unit to span multiple sessions or for several (rather short) units to be covered in a single session. The set of teaching materials required for a course can vary quite a lot. At the OSG, there are a number of typical teaching scenarios:

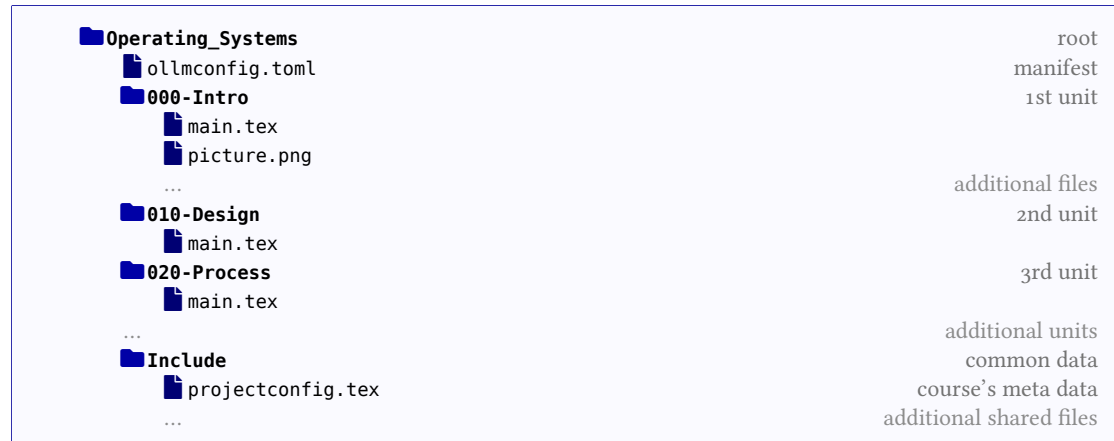
1. There are presentations that lecturers use whilst students are given the relevant handouts.
2. Same as 1, but the handouts are available in both English and German.
3. As in 2, but instead of handouts for each teaching topic, there is one comprehensive set of lecture notes for each language.

All of these scenarios—and many more—are supported by **OSG Lecture**. In this context, **osglecture** acts as a *metaclass*: It essentially determines which specific class is loaded (e.g., **article**, **book**, **ltx-talk**, or **beamer**) and ensures that there is a common interface. Materials that are not as directly related to the presentation itself or to handouts—such as lecture notes or textbooks—are grouped under the term “long form”.

3.3 Directory Structure

The materials for each course are located in their own directory, known as the (course) project directory or (in relation to the project) root directory. The directory structure mirrors the structure of the course.

The following shows a possible directory structure for a simple scenario:



The directory name of a course unit consists of at least a three-digit number and a description of the topic, separated by a hyphen. The number must be three digits long (leading zeros are allowed) and simply determines the order of the learning units, not chapter numbers. Therefore, at least during the development phase of a course, it makes sense to choose a larger increment so that chapters can be reorganized later without much effort.¹⁶

The topic description in the directory name can be chosen freely and is intended only as a reminder of what the content of the respective unit is. Multiple words can be separated by an underscore or a dash, but as a general rule, it is recommended to use a single, meaningful word.

However, the directory name may also contain additional information. Overall, the directory name of a course unit has the following format:

$$\langle number \rangle [\langle target \rangle] - [\langle role \rangle] - \langle topic \rangle$$

The parts in square brackets are optional. They have the following meanings:

$\langle target \rangle$ Beschreibt das Ziel dieser Einheit. Sometimes there is material that is only relevant for certain types of materials. For example, the organizational details of the introductory lecture are often not included in the lecture notes. This can be described using the "Target(scope)." It is a code consisting of one or two letters:

- a The directory contains only material for long-form work (script, book, etc.).
- b The directory contains materials for a presentation.

¹⁶ Older programmers may still be familiar with this principle from programming languages such as BASIC.

An additional second letter can narrow it down even further. For example, “bh” stands for handouts only, so no slides are created.

Without a target specification, the directory contains material for *all* configured document types.

This target range can be extended; see the documentation for `ollm`.

⟨role⟩ The role that the unit plays. This is always described by a single letter

- a Unit is an appendix.
- c This is synonymous with an omitted role specification and represents a normal unit.
- e The unit is a supplement. This means that it is related to the preceding unit but has a certain degree of independence. For example, it could be a worksheet or supplementary material.
- i This unit integrates other units. For example, the integration unit for a lecture transcript would combine all the transcript chapters (the other units) and add a title page, a common table of contents, etc. Each document type may have only one integration unit.

Attention

Please note that the structure of the directory names is not merely a convention: both the main class `osglecture` and the tool `OLLM` interpret these names.

Each course-unit directory contains a L^AT_EX file. It uses the document class `osglecture` and, in the default configuration, is named `main.tex`. The directory may also contain any other material, such as images, files to be included, or private `.sty` files needed only by that unit.

By contrast, private `.sty` files or other material needed by *all* course units should be placed in the `Include` directory.

3.4 Modi

Anyone who has worked with `beamer` or `ltx-talk` will know modes and overlay specifications. Both allow different presentations to be typeset from the same source, but they answer different questions.

For readers new to these concepts: both may appear, even together, in angle brackets as the first argument of a command. A *mode* decides during compilation whether the command and therefore its content is included. Text intended for slides but not, for example, for the lecture notes can thus be marked accordingly. Overlays can produce several PDF pages from one designed slide. They differ slightly but carry the same slide number. This supports effects such as revealing or highlighting content step by step. Suppose a slide initially shows only a question and reveals the answer after the next click. That is an overlay: both views belong to the same

PDF, target, and language version. If the detailed reasoning should instead appear only in the lecture notes and be absent from slides altogether, that is a mode decision. It produces different content in different document instances.

Modes and overlays are sometimes confused, partly because [beamer](#) and [ltx-talk](#) often combine them and use related syntax. Table 1 summarizes the distinction.

Table 1: Overlay und Modus

Question	Mechanism	Example
Does the content belong in this output?	mode	longform, presentation
When does it appear within the presentation?	Overlay	<2->

The [osglecture-modes](#) package supplies the common mode system, which can be used independently of [beamer](#) and [ltx-talk](#). The presentation classes' own modes remain available, but should only be queried when a feature specific to that class is actually intended.

The confusion is reinforced by syntax: presentation classes often write overlay specifications in angle brackets, and [OSG Lecture](#) also uses angle brackets for mode arguments. The command determines which concept is meant. With `\lecturemode<longform>` it is a mode. With an `\item<2->` supplied by the base class it is an overlay. An overlay specification is not a portable mode condition.

Modes in [OSG Lecture](#) are not a flat collection of switches but a directed graph from specific to general. For example, the concrete handout mode has the parents presentation and print; script leads through article to longform and print. Querying a parent mode therefore matches all appropriate specializations. This is why authors should normally express their intent as presentation, longform, slides, or print instead of testing for a base class.

Target, document type, and profile act in sequence. The target selects the requested output, the document type activates its semantic leaf mode, and the profile adds technical modes belonging to the typesetting environment. For example, the [beamer](#) profile adds beamer; another presentation profile need not have that technical mode. The parent graph nevertheless ensures that both profiles satisfy the same portable presentation query. Profiles may also supply additional modes and a mode setup file. They can thus extend the capabilities of a typesetting environment without replacing the meaning of the document type.

Overlays, by contrast, remain the responsibility of the selected presentation class. [OSG Lecture](#) can integrate common overlay commands, but it does not promise that every class- or theme-specific animation survives a profile switch. Authors requiring portability between [beamer](#) and [ltx-talk](#) should use the common structural vocabulary and restrict overlays to the subset supported by both profiles. Concrete content selection continues in Section 4.5.

Extensibility is useful when a project needs a semantically new output, not merely a different layout. For example, supervised self-study might introduce a studyguide type with its own active modes and a long-form profile. Merely switching from [book](#) to [scrbook](#), by contrast, only requires another profile for the existing script type. This distinction prevents technical typesetting decisions from needlessly creating additional document instances. The complete

profile example is in Section 7; new OLLM targets are registered using the target definitions described there.

3.5 Document Types and Profiles

A *document type* (*doctype*) denotes the conceptual output form you request, for example slides, handout, script, or article. Think of the question “What should be produced, and for whom?” rather than of a particular L^AT_EX class.

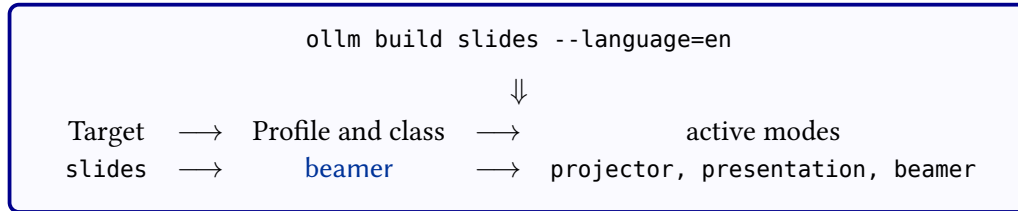


Figure 7: From the user’s choice to the concrete rendering.

In the normal workflow, the document type appears as the target name in the `OLLM` command. `OLLM` checks the manifest to determine whether this target and the requested language are configured for the project. It then passes the normalized selection to `osglecture`.

The subsequent build *instantiates* the shared source for precisely this selection. The concrete result is called a *document instance*. Thus, two instances may contain the same subject matter while differing in language, document type, or both.

For each target, the class selects a *profile*. The profile determines the concrete base class and its adapter. Thus, slides can use `beamer` in one project and `ltx-talk` in another without rewriting the course units. Target-specific exceptions are possible as well, for example `beamer` for slides and `ltx-talk` for handouts.

The *base class* is the actual L^AT_EX class that performs the typesetting, such as `beamer`, `ltx-talk`, `book`, or `scrbook`. An *adapter* translates the uniform `osglecture` interface into the commands and conventions of that base class. The subject-matter source can therefore remain independent of the selected base class.

Select the project-wide presentation profile with `presentation-profile` and the long-form profile with `longform-profile` in `Include/projectconfig.tex`. To override the profile for just one target, use :

```
\LectureTargetSetup{<target>}{profile=<profile>}
```

In standalone mode (see Section 6), use the `profile` class option instead. Choosing a profile affects more than appearance. For example, `beamer` provides a large ecosystem of existing themes, whereas `ltx-talk` is designed around Tagged PDF and automatic structure tagging. For long forms, `book` and `scrbook` differ in their class options and customization facilities, among

other things. A profile change should therefore be checked with a full build of the affected targets.

Document types and classes are therefore deliberately separate. This separation keeps your sources stable: in the project you decide *how* slides are typeset; in the course unit you only write *what* should appear in slides and long forms.

slides screen-oriented presentation for teaching;

handout print-oriented presentation version for the audience;

screen the presenter's version of the slides: every page is twice as wide and shows, to the right of the slide, optionally the current and the next slide, the remarks recorded with `\Note`, and a clock. With an extended desktop the left half goes to the projector;

script continuous long form, typically a chapter or complete set of lecture notes;

article standalone long form, for example a paper.

Some profiles also differ in whether they require L^AT_EX's new PDF management. In particular, `ltx-talk` requires a `\DocumentMetadata` call that technically has to precede `\documentclass`. In a series, the unit cannot place the appropriate call first itself because language and target are only known from the build request. The early call therefore belongs in `Include/documentmetadata.tex`:

```
\DocumentMetadata{
  lang=\OsgLectureRequestedLanguage,
  tagging=on
}
```

This is deliberately separate from visible information such as `\title` and `\author`: `documentmetadata.tex` activates PDF infrastructure, whereas `projectconfig.tex` describes the project's subject-matter and design context.

This list is extensible. When starting out, however, use the supplied types: they are already registered with OLLM and have suitable profiles, modes, and scopes.

Profile	Base class	Document types	Characteristics
<code>beamer</code>	<code>beamer</code>	slides, handout, screen	large choice of themes
<code>ltx-talk</code>	<code>ltx-talk</code>	slides, handout, screen	Tagged PDF required
<code>book</code>	<code>book</code>	script, article	standard class
<code>scrbook</code>	<code>scrbook</code>	script, article	KOMA-Script

Table 2: Document profiles supplied with the bundle.

4 To Shape Content

4.1 Titles and Metadata

Even though each teaching unit is *physically* located in its own directory, there is a command for the *logical* structure:

```
\lecture[⟨short name⟩]{⟨name⟩}{⟨Label⟩}
```

Starts a logically named teaching unit and creates its title slide in presentation forms.

In presentation forms, `\lecture` creates a title slide that normally repeats the course-unit title and relevant metadata. Its exact appearance depends on the presentation theme, if one is used.

All course units in a project belong to the same course and therefore share the course name, the author or authors, and possibly further metadata. To avoid repeating these for every unit, store them in `Include/projectconfig.tex`. The following commands are available:

```
\course<⟨mode⟩>[⟨short name⟩]{⟨name⟩}
```

Sets the course name for all modes or for the specified mode.

```
\event<⟨mode⟩>[⟨short name⟩]{⟨name⟩}
```

Sets the event name for all modes or for the specified mode.

The commands `\title`, `\subtitle`, `\author`, `\institute`, and `\date` have the same signatures. Through `course-target` and `event-target`, the profile decides whether course and event metadata are mapped to the title, subtitle, or neither. The complete setters and query commands are listed in Section 7.7.

The three argument forms are not interchangeable. Angle brackets select a mode, square brackets contain the short text, and braces contain the required long text. For example:

```
\course<presentation>[OS]{Betriebssysteme}
\course<longform>[Betriebssysteme]{Betriebssysteme und Systemsoftware}
```

sets different short and long values depending on the document instance. Without a mode, the value applies to all instances; without a short form, the long value is also used as the short value.

4.2 Presentations

When developing presentations, **OSG Lecture** provides the common structure formed by `\frame`, `\frametitle`, and `\lecture`, regardless of the selected profile. Overlays and theme-specific commands, however, come from the selected base class. A theme designed for **beamer** is therefore not automatically usable with **ltx-talk**. Use the modes described in the following section for portable content selection.

By default, `patch-overlays=true` integrates expressions made up of registered modes into the native mode and overlay commands of **beamer** and **ltx-talk**. A specification such as `<longform>` or `<slides||handout>` is then evaluated portably; for `<slides:2->` **OSG Lecture** evaluates the mode part and hands the presentation class only the overlay range `<2->`. A purely numeric specification such as `<2->` remains a class-specific overlay. `|` and `,` inside `<...>` always belong to the presentation class's syntax; `||` is the portable disjunction operator. Unknown expressions are delegated unchanged to the base class. Set `patch-overlays=false` to disable this integration; `\lecturemode` and `\IfLectureModeTF` continue to work.

4.3 Remarks and Source Credits

The command `\Note{<remark>}` records a remark for the presenter. In a document instance of type screen, remarks appear beside the slide; they are not typeset on audience slides or in long forms. Place the command inside the frame to which the remark belongs.

```
\begin{frame}
  \frametitle{Virtueller Speicher}
  Seitentabellen ordnen virtuelle Seiten physischen Seitenrahmen zu.
  \Note{Hier die Abbildung schrittweise erläutern.}
\end{frame}
```

Source credits, for example directly below a figure, are typeset with `\sourcederef`. Its optional argument contains the label and its mandatory argument the source. The language-dependent default label is “Quelle” or “Source”; an empty optional argument suppresses it. The starred form produces a separate, flush-right source line. In a tagged document, the source credit is regular content and remains part of the reading order.

```
\includegraphics[width=.7\linewidth]{page-table}
\sourcederef{Eigene Darstellung}
```

For a remark beside the main text flow, use `\osgannotation{<content>}`. Its argument will typically contain a handwritten annotation placed over the slide with TikZ. Put the call at the position in the source where the annotation should be read: in tagged output, this position, rather than its drawn position on the page, determines the reading order.

4.4 Multilingual Documents

In an *OSG Lecture* project, OLLM and the package *langselect* share the work: the manifest states *which* language versions may be built; OLLM selects one of them; *langselect* typesets the matching wording in the source. This division of responsibility keeps build logic out of the subject matter.

First list every project language in the manifest (`[targets.defaults].languages`, in a fixed order), then map it in *Include/projectconfig.tex* to a name understood by *babel*. For German and British English, a practical configuration is:

```
\LectureProjectSetup{
  languages={
    map={de=ngerman,en=british},
    load babel
  }
}
```

The selectable languages themselves are no longer repeated here – *OSG Lecture* takes them from the manifest.

Without an explicit *targetlang* sub-key, *OSG Lecture* automatically uses `\osgLectureLanguage`, that is, the project language selected by the build. *targetlang* remains available as an explicit override.

In the course units, use the provided language commands. With `selectable = {de,en}`, `\ldeen{Deutsch}{English}` is the natural form. Write short, complete variants; punctuation belongs inside each language version whenever it may differ.

```
\section{\ldeen{Synchronisation}{Synchronization}}
\ldeen{Ein Semaphor schützt den kritischen Abschnitt.}
      {A semaphore protects the critical section.}
```

Do not duplicate values, formulae, commands, or references that are identical in both languages. Use the starred form and placeholders instead. A value changed later then still occurs in only one place:

```
\ldeen*{Der Scheduler wählt alle @1 eine Aufgabe aus.}
      {The scheduler selects a task every @1.}
      {$10\,\mathrm{ms}$}
```

The placeholders $\langle @1 \rangle$, $\langle @2 \rangle$, and so on are replaced in every language version by the same following arguments. This is particularly useful for technical terms, cross-references, and numbers. If a language requires a different word order, simply put the same placeholder in the appropriate position.

Attention

Start builds through OLLM and select the language with `--language`. **Do not** additionally hard-code the target language in the source; otherwise the manifest, output file name, and actually typeset language can disagree.

The most reliable way to check whether a translation is complete is to typeset every intended document instance of the unit. A successful German version does not tell you whether an argument or command is missing in the English branch.

```
ollm build --all
```

4.5 Selecting Content with OSG Lecture Modes

Section 3.4 introduced the conceptual distinction between modes and overlays and their interaction with targets and profiles. This section covers the concrete author interface: modes select content and metadata for a document instance; they do not control the click sequence within a presentation. A *mode* is a named property of the document instance currently being created. It answers a simple question such as “Is this a presentation?”, “Is this a long form?”, or “Is this a print output?”. The command `\lecturemode<mode>{<content>}` emits its content only when the current instance has that property.

Users familiar with `beamer` or `ltx-talk` will already know similarly written mode commands. `\lecturemode` is OSG Lecture’s portable interface: it queries the mode graph independently of the document class and profile and therefore continues to work when switching between the two presentation classes as well as in long forms. Here the angle brackets select a mode; they are not a Beamer overlay specification.

When adapting content, begin with the most general intention. If a paragraph should appear only in a long form, select the `longform` behavior mode; if it belongs only on slides, select `slides`. Selecting the concrete base class is rarely necessary and makes a later profile change harder.

OSG Lecture distinguishes selected leaf modes from their active parent closure. `\LectureModes` lists the modes selected directly; `\ActiveLectureModes` additionally contains every more general mode reachable through the mode graph. For Beamer slides, for example, the first list may contain `projector,beamer`, while the second also activates `presentation`. Here, `slides` is the user-facing alias for `projector`. Content selection almost always means the parent closure, which is precisely what `\lecturemode` tests.

```

\lecturemode<longform>{
  \ldeen{Ausführliche Herleitung der Laufzeit:  $T(n)=\Theta(n\log n)$ .}
    {Detailed derivation of the running time:  $T(n)=\Theta(n\log n)$ .}
}
\lecturemode<slides>{
  \centering  $T(n)=\Theta(n\log n)$ 
}

```

Modes are properties of the output in this context. A concrete document type also activates more general modes: a handout, for example, is both a presentation form and a print output. Thus, the same content can be enabled with `<presentation>` for slides and handouts, or with `<print>` for all print-oriented results. Multiple alternatives can be joined with `||`.

The same mode argument is available on metadata commands such as `\course<⟨longform⟩>` and on conditionally loaded preamble fragments. The selection rule therefore remains the same at every level: the mode graph is first completed from document type and profile, after which mode-dependent values and content are evaluated. Custom modes must consequently be declared before this finalization; the project configuration is described in Section 7.6.

4.6 Lists as Continuous Text

For lists that read better as prose in lecture notes, `presitemize` is the more convenient choice. Entries describe statements; their introductory commands specify the logical relationship between them.

```

\begin{presitemize}
  \item Die Übersetzung \finite{funktioniert}
  \anditem das Programm \finite{liefert} ein Ergebnis
  \butitem wir \finite{haben} vereinfacht
  \thereforeitem wir \finite{müssen} das Ergebnis prüfen
\end{presitemize}

```

In presentations this remains a list. In the long form, `presitemize` combines the statements approximately as follows: “The translation works and the program produces a result, but we simplified, and therefore we have to check the result.” `\finite` marks the finite verb so that it can be moved to the correct position, especially in a German subordinate clause.

For punctuation-only continuations, `\commaitem` and `\colonitem` are available. In long-form output they produce “, ” and “: ”, respectively, without an additional conjunction; in presentations they appear as ordinary items. The equivalent key forms are `\presitem[relation=comma]` and `\presitem[relation=colon]`.

Use `presitemize` for flat sequences of short statements. For nested lists, tables, or substantially different explanations, explicit mode variants are clearer. As a rule of thumb: if only the

linguistic connection changes, `presitemize` is sufficient; if the didactic structure changes, use `\lecturemode`.

Hint

The automatic prose joining performed by `presitemize` currently has language profiles for German, English, and French, including common variants such as `ngerman`, `british`, and `fr-CA`. For another language the package issues a warning and uses the English profile for its connectives as a fallback. This is useful only if the affected output actually contains English text.

For additional languages, write the long form explicitly and use `\lecturemode` to select between it and the presentation list. This keeps word order, inflection, and punctuation under your control; the individual formulations can still be selected with the language commands generated by `langselect`.

4.7 Two-Column Content

What sits side by side on a wide slide, such as a figure and its explanation, usually reads better one after the other on the narrower page of a long form. The `twocolumns` environment makes this distinction itself: it typesets two parts separated by `\nextcolumn` side by side when a mode expression matches, and consecutively otherwise. Without one, the mode expression is presentation.

```
\begin{twocolumns}[t,.46]
  \includegraphics[width=\linewidth]{page-table}
  \nextcolumn
  Seitentabellen ordnen virtuelle Seiten physischen Seitenrahmen zu.
\end{twocolumns}
```

In the optional argument, a number between zero and one sets the first column's share of the available width (default `.5`). One value `t`, `c`, or `b` aligns both columns at the top, centre, or bottom; two such values align the columns individually. `sep` sets the gutter between the columns, for example `sep=1cm`, replacing the default `\columnsep`. The mode expression precedes the optional argument in angle brackets: `<all>` forces columns in every mode, `<longform>` only in long forms, and `<presentation|longform>` in both behaviour classes.

4.8 Cross References

For references within the same document instance, you can use the usual `hyperref` commands. In addition, `OSG Lecture` provides `\olref` and `\olpageref`. Without an optional document address they behave like `\ref` and `\pageref`:

```
\section{Scheduling}\label{sec:scheduling}
Siehe Abschnitt~\olref{sec:scheduling} auf Seite~\olpageref{sec:scheduling}.
```

The optional address before the label names another logical unit in the same series. As explained above, this logical identity is set by the third argument of the `\lecture` command. By default, the reference uses the same document type and language as the current document. Thus, from the memory unit, the following reference leads from a German script to the German script of processes, but from English slides to its English slides:

```
\olref[processes]{sec:scheduling}
```

The `type` and `lang` keys explicitly override that document instance. This creates a cross-document-type reference, for example from slides to the detailed derivation in the script:

```
\olautoref[processes,type=script,lang=de]{sec:scheduling-proof}
```

A keyless address entry is always the unit ID. Use stable logical IDs rather than directory names; OLLM resolves the requested combination of unit, document type, and language through its reference snapshot. The target document must have been built successfully at least once; only then are its references available to other units. After changing dependent units, `ollm build --resolve` is the easiest way to rebuild producers and consumers to a consistent state.

4.9 Literature

For literature references, *OSG Lecture* uses *biblatex* with *biber*. Bibliography management is enabled in `Include/projectconfig.tex`; the database usually lives in `Include` as well, where *OLL*M makes it available to *biber*:

```
\LectureProjectSetup{
  bibliography = {resources = lecture.bib}
}
```

The default style `alphabetic` produces labels such as [Knu69]; `style` selects another *biblatex* style, and `options` passes further load options on. Cite with the usual commands such as `\cite`. `\printbibliography` prints a bibliography; which document types receive one is controlled through modes like everything else, for example only in the script:

```
\mode<longform>{\printbibliography}
```

A citation links to the bibliography only if it actually prints the entry in the same document. In slides without a bibliography, the label therefore remains plain text rather than a link to nowhere. Bibliographies by topic or chapter are available through the `biblatex` filters, such as `keyword=`, `type=`, or `category=`.

A series-wide bibliography is printed by `\printseriesbibliography`: all entries cited by any unit of the series with the same document type and language, filtered as with `\printbibliography`. It can stand in any unit, for example in a literature unit of its own or at the end of a chapter. If a unit cites something new, `ollm check` reports the bibliography as stale; `ollm build --resolve` rebuilds it. If the project configuration names this unit as `series-target`, citations in other units link to its entries whenever the citing document has no bibliography of its own with the entry; on integration these links become internal:

```
\LectureProjectSetup{
  bibliography = {resources = lecture.bib, series-target = literature}
}
```

An annotated reading list shows selected entries in full, organized in groups:

```
\begin{literaturelist}
  \litgroup{Grundlagen}
  \litentry{Knu:Art-1:1969}
  \litcomment{Das Standardwerk; umfassend, aber anspruchsvoll.}
  \litgroup{Entwurf von Algorithmen}
  \litentry{Ski:AlgorithmDesign:2008}
\end{literaturelist}
```

`\litcomment` comments on the preceding entries; an optional argument puts a label in front of the comment. The optional argument of the environment sets the width of the label column.

4.10 Command Reference

The following summaries list the complete public interface for the tasks covered in this chapter. Commands for defining new profiles or adapters are instead listed in Section 7.7; they are not needed when writing ordinary course units.

Course unit and metadata.

```
\lecture[⟨short title⟩]{⟨title⟩}{⟨unit ID⟩}
  starts the logically identified course unit.

\course<⟨mode⟩>[⟨short value⟩]{⟨long value⟩}
```

`\event<mode>[short value]{long value}`
`\title<mode>[short value]{long value}`
`\subtitle<mode>[short value]{long value}`
`\author<mode>[short value]{long value}`
`\institute<mode>[short value]{long value}`
`\date<mode>[short value]{long value}`
 set a metadata field; mode and short form are optional.
`\OsgLectureSetField<mode>[short value]{long value}`
 uniform setter name for every predefined or declared metadata field.
`\OsgLectureField`
`\OsgLectureShortField`
 expandable query of the long or short value, respectively.

Modes and conditional content.

`\lecturemode<mode expression>{content}`
 emits the content if at least one mode separated by a comma or || is active; without a mode expression it always emits the content.
`\IfLectureModeTF{expression}{true}{false}`
 full conditional.
`\IfLectureModeT{expression}{true}`
 true branch only.
`\IfLectureModeF{expression}{false}`
 false branch only.
`\LectureModes`
`\ActiveLectureModes`
 expandable comma-separated lists of the selected leaf modes and their complete parent closure, respectively.

The commands for extending the mode graph are summarized in Section 7.6; they belong to project and profile configuration rather than ordinary unit authoring.

Presentation lists.

`\begin{presitemize}[\langle language, punctuation, whitespace \rangle]`

produces a list in presentations and a grammatically joined paragraph in long forms. `language` overrides the detected language; `punctuation` accepts auto or manual and preserves existing punctuation in auto mode; `whitespace` accepts normalize (the default) or preserve.

`\presitem[\langle relation, initial \rangle] <\langle overlay \rangle> [\langle label \rangle]`

`relation`: sentence, and, but, therefore, comma, colon; `initial`: auto, keep.

`\anditem <\langle overlay \rangle> [\langle label \rangle]`

`\butitem <\langle overlay \rangle> [\langle label \rangle]`

`\thereforeitem <\langle overlay \rangle> [\langle label \rangle]`

`\commaitem <\langle overlay \rangle> [\langle label \rangle]`

`\colonitem <\langle overlay \rangle> [\langle label \rangle]`

short forms for the corresponding relations; each accepts an overlay specification and an optional label like `\item`.

`\finite{\langle text \rangle}`

marks the finite part of a subordinately joined clause.

`\prespar`

starts a new paragraph in long-form output and closes the running sentence; no effect in presentations.

`\longform`

`\slides`

`\longslides`

`\prescase`

`\preskeepcase`

low-level selection and capitalization helpers for custom list macros.

Two-column content.

`\begin{twocolumns} <\langle mode expression \rangle> [\langle alignment, share, sep=length \rangle]`

typesets two parts side by side when the mode expression matches (default presentation), and consecutively otherwise.

`\nextcolumn`

separates the two parts of twocolumns.

References.

`\olref*[\langle address \rangle]{\langle label \rangle}`
 number like `\ref`.

`\olpageref*[\langle address \rangle]{\langle label \rangle}`
 page like `\pageref`.

`\olnameref*[\langle address \rangle]{\langle label \rangle}`
 name like `\nameref`.

`\olautoref*[\langle address \rangle]{\langle label \rangle}`
 automatic reference like `\autoref`.

The address consists of a keyless unit ID and optional `type = {doctype}` and `lang = {language}`. Without an address, all four commands remain local. As with the corresponding standard commands, the starred form suppresses the hyperlink.

Literature.

`\begin{literaturelist}[\langle label width \rangle]`
 annotated reading list; available when bibliography management is enabled.

`\printseriesbibliography[\langle options \rangle]`
 series-wide bibliography; `\langle options \rangle` as for `\printbibliography`.

`\litentry{\langle key \rangle}`
 full entry with citation label.

`\litgroup{\langle title \rangle}`
 starts a group with a heading.

`\litcomment[\langle label \rangle]{\langle text \rangle}`
 comment on the preceding entries.

Citation and bibliography commands are those of `biblatex`, such as `\cite` and `\printbibliography`.

5 Series

A series is more than a set of directories. Its units may depend on one another: chapter numbers should continue, a cross-reference needs a target from another unit, and a combined document needs the built individual chapters. **OLL**M keeps the state of successful builds for these purposes and can rebuild stale producers with `--resolve`. Without this series state, a local L^AT_EX run could only see the current source file.

5.1 Continuing Counters

Each unit is built as an independent document. This enables separate distribution and incremental builds, but initially creates another problem: L^AT_EX would restart chapters, sections, and pages at one in every unit. A subsequently integrated set of notes would then contain several Chapter 1s. Counter continuation therefore imports selected final values from the preceding unit applicable to the document type.

Which counters should continue is a rendering decision. Slides usually do not need page numbers that run across units, whereas lecture notes often do. The setting therefore belongs to the longform mode in [Include/projectconfig.tex](#):

```
\0sgLectureSetup{
  mode/longform={
    continuation={counters={page,chapter,section}}
  }
}
```

Suppose 010-introduction ends with Section 2. The first `\section` call in 020-processes then produces Section 3. A counter not listed, such as figure, still restarts. This explicit selection matters: local counters should not accidentally acquire meaning across unit boundaries. To disable continuation entirely for a long form, use `continuation=none`.

The preceding instance must have been built successfully at least once so that its counter state is available. After inserting a new section in an early unit, the following command updates the dependent German script instances:

```
ollm build script --language=de --scope=series --resolve
```

5.2 Integration

The documents produced for the course units in a series can be used individually, as is usually appropriate for presentations. They can also be integrated into a single document. Each course unit might, for example, form a chapter that is integrated into a complete set of lecture notes; a collection of handouts can be assembled in the same way. In [OSG Lecture](#), a complete set of lecture notes is normally not created by applying `\include` to the TeX sources, but by combining documents that have already been compiled. This requires a separate integration unit containing a wrapper document. Its role element is marked with the letter “i”. The directory’s sequence number does not affect numbering. It is nevertheless advisable to use either the smallest or the largest number so that all units to be integrated can remain adjacent.

```
course/
  ollmconfig.toml
  Include/
    projectconfig.tex
    documentmetadata.tex
  010-introduction/main.tex
  020-processes/main.tex
  900-i-collection/main.tex
```

The normal integration workflow is incremental: while writing, first build the documents to be integrated and build the wrapper document last. Before building the wrapper, use `ollm check` to verify that all documents to be integrated have been created and all references have been resolved. Further compilation runs may be necessary. OLLM can help here as well: the `--resolve` option discovers and triggers dependencies up to a specified depth.

```
cd 020-processes
ollm script --language=de
ollm script --language=de --scope=series --resolve
ollm script --language=de --scope=collection --resolve
ollm check --scope=series
```

5.3 Wrapper Document

A complete script is not created by applying `\include` to the TeX sources. Instead, the wrapper document is a separate unit with role `i`. The class recognizes this role from the directory name and loads the integration service automatically. The document names the logical units to include in the desired order:

```

\documentclass{osglecture}

\begin{document}
\includeunit{introduction}
\includeunit{processes}
\includeunit{synchronization}
\end{document}

```

`\includeunit` automatically selects the same language and document type as the wrapper. Thus a German script imports exactly the German script instances. OLLM builds the collection with `--scope = {collection}`; if there is no matching integration unit for the document type, or more than one, the selection is ambiguous and the build fails.

Every document instance to be included must have been built at least once. The integration run records the build actually used as a dependency. After changes, another run with `--resolve` is therefore normally sufficient. The order of the `\includeunit` commands is the order in the result; it is not inferred from directory numbers.

5.4 Tags, Links, and Fallback

If the wrapper and imported units are built as Tagged PDF, `tagpax` carries across pages, the structure tree, destinations, links, table-of-contents entries, and bookmarks. This preserves accessibility structure and navigation across unit boundaries. The target must permit or require `\DocumentMetadata`, and `documentmetadata.tex` must enable tagging.

```

\DocumentMetadata{
  lang=\OsgLectureRequestedLanguage,
  tagging=on
}

```

For untagged document instances, integration uses a fallback based on `pdfpages` and `newpax`. Links and document navigation largely remain functional, but imported pages are included as an artifact and have no accessible reading order or heading structure. Mixing tagged and untagged units forces the untagged path for the affected document type and produces a prominent warning. A consistently tagged series is therefore both higher quality and more predictable.

5.5 Reference Summary

`\includeunit{⟨unit ID⟩}`
includes, in an integration unit, the already built instance with the same language and document type; call order determines document order.

Integration is available only in a series with exactly one matching unit of role `i` per document type. It requires successfully built source documents. Cross-unit references, counter continuation, and integration are resolved through OLLM's state and dependency model; local LaTeX runs without that state cannot reconstruct them.

6 Standalone

In addition to the series workflow considered so far, **OSG Lecture** provides a second mode of operation: standalone mode. It typesets a single document using an **OSG Lecture** profile without requiring a course manifest and a series of unit directories. A typical use case is a slide deck for a talk at a scientific conference.

In standalone operation, the document describes its own output and is typeset without a project manifest. The subject-facing commands remain largely the same; the main difference is where context and defaults come from.

A standalone document has neither a build request nor a series manifest and must declare this explicitly with the **standalone** class option. Alternatively, **OLLM** supports the convenience command `ollm standalone main.tex`. Using the class mode directly is usually simpler: without a set of automatically generated document instances, series dependencies, or deployment, there is little for **OLLM** to coordinate, and `latexmk main.tex` is sufficient.

Hint

The choice depends on the problem, not on document length: a four-page handout built in three languages alongside twelve course units belongs in a series. A one-off hundred-page conference volume without neighbouring documents may still be standalone.

The document type and language that **OLLM** supplies in a series are now selected with **doctype** and **lang**. The document profile can be overridden with **profile**:

```
\documentclass[
  standalone,
  doctype=script,
  profile=scrbook,
  lang=de,
  class-options={11pt,oneside}
]{osglecture}
```

In this example, `script` is the semantic output form, `scrbook` is the technical typesetting class, and `de` is the language selected directly by the document. In a series these three decisions would come from the manifest and project configuration; standalone mode deliberately makes them visible in the document.

doctype defaults to `slides`, and built-in document types have default profiles. Explicit values nevertheless make a standalone document clearer and more reproducible. **class-options** passes options directly to the base class selected by the profile. Having both an **OLLM** build request and the **standalone** option is an error because two sources would otherwise determine the same build context.

Without a manifest, `\include/projectconfig.tex` is not discovered automatically. Shared packages, language configuration, title, and custom commands must therefore be loaded in the document or explicitly read from a file. A standalone multilingual script might begin as follows:

```
\documentclass[standalone,doctype=script,profile=scrbook,lang=de]
  {osglecture}
\usepackage[
  languages={de,en},
  targetlang=de,
  map={de=ngerman,en=british},
  load babel
]{langselect}
\title{\ldeeen{Betriebssysteme}{Operating Systems}}
```

Profiles such as `ltx-talk` require `\DocumentMetadata` before `\documentclass`. A standalone document has no `documentmetadata.tex` inserted by OLLM, so the call must genuinely be placed at the start of the file:

```
\DocumentMetadata{lang=en,tagging=on}
\documentclass[
  standalone,doctype=slides,profile=ltx-talk,lang=en
]{osglecture}
```

Such a document can be built directly with Lua^AT_EX or latexmk. The compatible convenience command `ollm standalone main.tex` also uses the class options; OLLM target, language, and matrix options are not permitted in this mode. Without series state there are no external unit references, counter continuation, or `\includeunit` resolution. Local references and all mode-aware rendering commands continue to work unchanged.

7 Project Configuration

The configuration of **OSG Lecture** is deliberately not kept in one file. Different layers answer different questions: **OLLM** decides *what* may be built and where results live; **L^AT_EX** decides *how* a selected document instance is typeset. Put a setting at the lowest layer at which it applies to all affected documents.

User configuration selects personal defaults, currently in particular the preferred bundle preset.

ollmconfig.toml defines the project's identity, document instances, security, resolution, and deployment.

Preset and target definitions give a target name its meaning: document type, profile class, metadata contract, and permitted unit scopes.

Include/projectconfig.tex contains the shared semantic **L^AT_EX** configuration, such as profiles, languages, metadata, and class options.

Include/documentmetadata.tex contains the early **\DocumentMetadata** call for targets requiring Tagged PDF or other early PDF properties.

Course unit may refine project-wide metadata for its own content; enforced project settings remain binding.

7.1 The Project Manifest

The file **ollmconfig.toml** marks the project root. **OLLM** searches upwards for it from the working directory. The manifest describes the document instances to be generated and contains no teaching content. A compact project with two languages and three targets might begin as follows:

```
schema = 2
bundle_preset = "OSG lecture/1"

[project]
id = "operating-systems"

[targets.defaults]
languages = ["de", "en"]
default_language = "de"

[targets.slides]
[targets.handout]
[targets.script]
```

`schema` identifies the file schema, not the bundle version. `project.id` is the persistent technical series ID and contributes to job names and document identities. Change it after the first build only deliberately.

`[targets.defaults]` holds what every target inherits unless it sets its own: `languages` lists all project languages in a fixed order and is also the validation anchor (a target's `languages` must be a subset); `default_language` is built when the CLI names none. An empty `[targets.slides]` inherits everything; an omitted target simply does not exist. Mapping `de` to `ngerman` or `en` to `british`, however, is a L^AT_EX concern and belongs in `projectconfig.tex` as the `map` option of `langselect` – the language list itself need not be repeated there.

A target may override `languages`, `default_language`, `profile`, and `document_metadata`. `profile` selects the document profile for that one target; `[targets.defaults]` may instead set `presentation_profile` and `longform_profile`, which apply according to the target's profile class. If neither is set, the bundle preset's profile defaults apply. `document_metadata` with `enabled` or `disabled` is a deliberate output choice and takes effect only where the resolved profile leaves the metadata contract open (supported); if it contradicts a profile with a forced contract, OLLM rejects the build before the run.

`[project]` may additionally set `tex_directory` and `tex_config` (default `Include` and `projectconfig.tex`). The directory name stays project-root-relative, `tex_config` is a file name only:

```
[project]
id = "operating-systems"
tex_directory = "Include"
tex_config = "projectconfig.tex"
```

Other manifest areas control `[security]` (`shell_escape`, `overwrite`), the maximum number of build `--resolve` rounds, and deployment.

7.2 BundlePreset

The bundle preset is a versioned contract between the project and the definitions installed with OLLM. `OSG lecture/1` selects the vocabulary and rules of this bundle generation. Do not confuse it with a document profile: the preset explains the build world to OLLM, whereas a profile later selects the concrete L^AT_EX class and rendering.

The supplied target definitions register `slides`, `handout`, `screen`, and `script`. A definition specifies `doctype`, `profile_class`, and `unit_scopes`. The preset also carries the profile defaults `presentation_profile` and `longform_profile`. Profile descriptions (`document-metadata`, `doctypes`, and more) are supplied under `definitions/profiles/`; project-local definitions can be added through `.ollmconfig.local.toml` and do not silently replace built-in ones with the same name.

A custom target is useful, for example, when students should receive a study guide with a didactically different structure in addition to the normal lecture notes. Nora first registers the search path in `.ollmconfig.local.toml`:

```
schema = 1

[definitions]
paths = ["Definitions"]
```

She then creates `Definitions/targets/studyguide.toml`:

```
schema = 1
kind = "target"
name = "studyguide"
version = "1.0"
doctype = "studyguide"
profile_class = "longform"
unit_scopes = ["a", "as"]
```

Only this definition gives the name `studyguide` a meaning: it is a long form and applies to the listed unit scopes. The manifest may then add `[targets.studyguide]`. L^AT_EX also needs a profile supporting the new document type; Nora places its profile description as `Definitions/profiles/<name>.toml` next to the `.def`. A new name alone is deliberately insufficient; build selection and typesetting must understand the same contract.

7.3 User Defaults

Users working on several projects with the same bundle can set a personal preferred preset. On Unix the file is normally found in `$XDG_CONFIG_HOME/ollm/config.toml` or `$HOME/.config/ollm/config.toml`, and on Windows in `%APPDATA%/ollm/config.toml`. `OLLN_USER_CONFIG` can select a different path:

```
schema = 1
bundle_preset = "OSG lecture/1"
```

An explicit `bundle_preset` in the project manifest takes precedence. The user file is therefore a convenience default, not a way to override reproducible project configuration unnoticed.

7.4 L^AT_EX Configuration

The shared L^AT_EX configuration is stored in `Include/projectconfig.tex` by default. It describes the project's subject-matter and design semantics – profile selection and the language list live in the manifest instead. A typical core is:

```
\LectureProjectSetup{
  languages={
    map={de=ngerman,en=british},
    load babel
  }
}

\OsgLectureSetup{
  class/<slides>={aspectratio=169},
  class/scrbook={twoside,open=right},
  mode/longform={continuation={counters={page,chapter,section}}}
}

\title{\ldeen{Betriebssysteme}{Operating Systems}}
\author{Nora Example}
```

For an [OLLM](#) build the selectable languages come from the manifest; languages here only carries the [langselect](#) mapping. [OSG Lecture](#) hands the language list (in manifest order) from the build file to [langselect](#).

`class/NAME` contains unchanged options for the actual base class and is evaluated before that class is loaded: `class/beamer` addresses the base class by name, `class/<slides>` addresses the mode (the *finalized* mode graph is ready before `\LoadClass`; a single mode name is checked against the active modes). `mode/NAME`, in contrast, contains settings for an `osglecture` output property and is applied from general to specific after the mode graph has been finalized.

Freely programmable preamble code should preferably be placed in separate fragments included with `\IncludeOsgLecturePreamble{<name>}`. They may contain package loads and custom commands and can be restricted by a mode expression or `class = {<class>}`. This keeps `projectconfig.tex` a readable collection of semantic project decisions.

Metadata. Fields such as title and author may have mode-dependent values. Custom fields are registered with `\DeclareOsgLectureMetadataField`. Normal precedence is fallback, profile, project, and finally unit. `\EnforceLectureProjectConfiguration` makes a project setting binding so that a unit cannot override it.

```

\author<presentation>[N. Example]{Nora Example}
\author<longform>[Example]{Dr. Nora Example}
\DeclareOsgLectureMetadataField{supervisor}
\supervisor<longform>{Ada Professor}

```

`documentmetadata.tex` has a different role: its `\DocumentMetadata` call must precede the document class and controls PDF properties rather than visible subject metadata. `OLLM` includes the file when the metadata policy derived from the profile resolves to enabled – always for a required profile, and for a supported profile when the manifest sets `document_metadata = {enabled}`. Symbols such as `\OsgLectureRequestedLanguage` are available there.

7.5 Document Profiles

A document profile connects a conceptual document type to a technical typesetting environment. It names the base class, class integration, supported document types, additional modes, class options, and the `\DocumentMetadata` contract. The supplied profiles are:

beamer for slides, handout, and screen; metadata before the class is forbidden (preset default for `presentation_profile`).

ltx-talk for slides, handout, and screen; `\DocumentMetadata` is required.

book for script and article, based on the standard `book` class.

scrbook for script and article, based on KOMA-Script; `\DocumentMetadata` is forbidden because KOMA-Script does not support it yet, so no Tagged PDF is produced (preset default for `longform_profile`).

The profile is chosen in the manifest (`profile` on the target, or `presentation_profile/longform_profile` in `[targets.defaults]`), not in `projectconfig.tex` – its `\DocumentMetadata` contract has to be fixed before the class runs at all. For a standalone document without a manifest the class option `profile = {\langle name \rangle}` selects it.

A custom profile is normally provided as `osglecture-profile-NAME.def` on the TeX search path. It may reuse an existing class integration; a new base class also requires a suitable integration. The three required entries are `backend`, `class`, and `doctypes`:

```

\ProvidesExplFile{osglecture-profile-mybook.def}{2026-08-28}{1.0}
  {Project book profile}
\DeclareOsgLectureProfile{mybook}{
  backend=scrbook,
  adapter=scrbook,
  class=scrbook,
  capabilities={longform,print},
  doctypes={script,article},

```

```

class-options={11pt},
document-metadata=supported,
course-target=title,
event-target=subtitle
}
\DeclareOsgLectureProfileClassOptions{mybook}{script}{open=right}

```

So that `OLL` knows the `\DocumentMetadata` contract before the run, a machine-readable profile description `osglecture-profile-mybook.toml` sits next to the `.def` (or under `Definitions/profiles/`):

```

schema = 1
kind = "profile"
name = "mybook"
version = "1.0"
profile_class = "longform"
document_metadata = "supported"
doctypes = ["script", "article"]

```

After installation or adding the shared directory to the TeX search path, `longform_profile = {mybook}` in `[targets.defaults]` is sufficient. `OSG Lecture` loads the `.def` automatically on first selection and checks whether the requested document type is supported. `definitions/profiles.t` guards that the `.def` and the `.toml` do not drift apart.

7.6 Extending the Mode Graph

A custom mode is appropriate when a new output property needs to be queried in the content. A new colour scheme is not a mode; it only changes presentation. A study guide containing additional self-assessment questions while also being a printable long form, by contrast, reasonably needs the studyguide leaf mode with the parents `longform` and `print`.

The declaration belongs in a mode setup file loaded by a profile before the graph is finalized:

```

% osglecture-mode-studyguide.tex
\DeclareLectureMode[parents={longform,print}]{studyguide}
\AssignLectureModeBehavior{studyguide}{longform}

```

The parents describe semantic membership: a query for `studyguide`, `longform`, or `print` is true for this instance. The behaviour class answers a different question: which concrete implementation should polymorphic commands use for this leaf mode? Assigning `longform` lets the new type reuse existing long-form implementations. Only genuinely new behaviour

should introduce a new behaviour class with `\DeclareModeBehaviorClass`; every command contract applying to that class must then be implemented completely.

The corresponding profile connects the semantic mode to a typesetting environment. Compared with the previous profile example, the new document type and setup file are added:

```
\DeclareOsgLectureProfile{studyguide-scrbook}{
  backend=scrbook,
  adapter=scrbook,
  class=scrbook,
  doctypes={studyguide},
  mode-setup-file=osglecture-mode-studyguide.tex,
  document-metadata=supported
}
```

The lifecycle is deliberately strict: modes, parent edges, and aliases are declared first; then `OSG Lecture` activates the leaf modes supplied by the document type and profile; finally it finalizes the graph. Only then are mode-dependent configuration and content evaluated. Declarations and changes after finalization are errors. This ensures that every subsequent query sees the same validated, acyclic graph.

A profile's `mode-setup-file` is read only for its document types. If shared source names a project mode – such as `\mode<solutions>` in a unit that is also built as script – the mode has to be known in every target, since an unknown mode is an error. For that, `\LectureProjectSetup{mode-setup-file=<file>}` in `projectconfig.tex` reads a file in every target, before the profile's. The mode still becomes active only where the document type or profile selects it. The example `examples/series-modern` uses this for an exercise sheet with a solutions version.

`\DeclareLectureModeParents{<mode>}{<parents>}` supplies or replaces parents separately from declaration; `\DeclareLectureModeAlias{<alias>}{<mode>}` provides an alternative name. `\SetLectureModes` and `\AddLectureModes` select leaf modes, while `\FinalizeLectureModes` closes the graph. In ordinary `osglecture` projects, the class and profile perform the last three steps; project sources should not pre-empt them.

7.7 Configuration Reference

Class options.

standalone enable manifest-free operation.

doctype = {name} select the document type; defaults to `slides` in standalone mode.

profile = {name} select a document profile explicitly (standalone; for `OLLM` builds it comes from the manifest).

lang = {tag} select the document language.

class-options = {list} pass options to the base class.

patch-overlays = {true|false} integrate portable modes into native overlay commands.

\LectureProjectSetup{<key-value list>}.

class-options shared base-class options.

mode-setup-file = {file} mode declarations read in every target, before the profile's mode file.

presentation-profile, longform-profile, identity-profile profiles for standalone documents without a manifest; for **OLLM** builds profile selection here is deprecated and comes from the manifest.

theme, numbering, references project-wide semantic policies.

bibliography = {{resources, style, options, series-target}} enables bibliography management with **biblatex** and **biber** (default style alphabetic).

languages embedded **langselect** configuration.

Under **languages**, the permitted keys are **targetlang, prefix, auto, trim, map, load babel, load polyglossia, and unified shorthands**. **selectable** comes from the manifest for an **OLLM** build (a value here is dropped with a warning) and is meant for standalone only. If **targetlang** is omitted, **OSG Lecture** automatically supplies **\OsgLectureLanguage**. An explicit value overrides this convention.

Other project commands.

\OsgLectureSetup{<areas>}

assigns base-class options addressed by class name (class/NAME) or active mode (class/<MODE>), and **osglecture** mode-dependent settings with mode/NAME.

\LectureTargetSetup{<target>}{<settings>}

accepts **filecolor**; **profile** is ignored for **OLLM** builds (profile selection in the manifest).

\IncludeOsgLecturePreamble*<(<modes>)>[<class=list>]{<name>}

registers a conditional preamble fragment; the starred form loads immediately and therefore permits no conditions.

\LectureProjectEnforce{<settings>}

sets project policies with enforced priority.

\EnforceLectureProjectConfiguration{<code>}

runs arbitrary configuration code with enforced priority.

`\DeclareOsgLectureMetadataField{<name>}`

declares a metadata field together with its setter and query commands.

The predefined metadata fields are `title`, `subtitle`, `author`, `institute`, `date`, `course`, and `event`. Each field has a setter `\OsgLectureSet{Field}`, a long-value command `\OsgLecture{Field}`, and a short-value command `\OsgLectureShort{Field}`. Setters accept `<mode>[<short value>]{<long value>}`.

Defining profiles. `\DeclareOsgLectureProfile{<name>}{<settings>}` accepts the complete set `engine`, `capabilities`, `backend`, `adapter`, `class`, `doctypes`, `class-options`, `setup-file`, `modes`, `mode-setup-file`, `course-target`, `event-target`, and `document-metadata`. The last three choices are `title|subtitle|none`, `title|subtitle|none`, and `supported|required|forbidden`, respectively. Required fields are `backend`, `class`, and `doctypes`; `engine` is currently `lualatex`. Add type-specific class options with `\DeclareOsgLectureProfileClassOptions{<profile>}{<doctype>}{<options>}`.

Configuring references. `\OsgLectureReferencesSetup{<settings>}` accepts `legacy = {true|false}` and `replace`. `replace` may contain a list of `ref`, `pageref`, `nameref`, `autoref`, or the aggregate value `all`, replacing the corresponding standard commands with their series-aware variants.

8 OLLM

OLLM is the executable build interface of the bundle. It reads the manifest, discovers units, normalizes the target and language, creates an isolated build request, and runs it through `latexmk` with `LuaLATEX`. Source files therefore do not need to parse job names or infer project state. The general command form is:

```
ollm [globale Optionen] [Aktion] [Target] [Aktionsoptionen]
```

If no action is given, `build` is assumed. The supplied targets `slides`, `handout`, `screen`, and `script` may be written as bare words; project-defined targets are selected unambiguously with `--target = {NAME}`:

```
ollm slides --language=de
ollm build --target=script --language=en
```

8.1 Building: From One Instance to a Series

For day-to-day work in a unit, a single build is usually sufficient. Without `--language`, OLLM uses the project default language if supported by the target, otherwise the target's first configured language. `--source` defaults to `main.tex`:

```
ollm build script --language=en
ollm build --target=slides --source=main.tex
```

The scope determines how far the selection reaches. `current` builds one document instance of the current unit. `unit` builds its permitted target/language matrix. `series` builds one document instance of every applicable regular unit, whereas `collection` builds the uniquely matching integration unit for the document type. The last two scopes also work from the project root:

```
ollm build --all --scope=unit
ollm handout --language=de --scope=series
ollm script --language=de --scope=collection
```

`--all` expands the matrix only within the selected scope; it does not automatically mean “the entire project”. `--rebuild` forces a full `latexmk` rebuild. `--dry-run` displays the normalized selection without starting `LATEX` and, together with `--format = {json}`, is useful for tools and CI.

Cross-unit references and integrations form dependencies. `--resolve` rebuilds stale producers and consumers in several rounds until a fixed point is reached. A logical unit that has never been built directly must first be built once in the required document instance so that OLLM can associate its ID with its directory.

```
ollm build script --language=de --resolve
```

8.2 Inspecting and Diagnosing

report, check, and doctor answer three different questions. report describes the known state: units, document instances, versions, output files, and dependencies. Findings do not change its exit status as long as the state is readable. check reads the same data but treats inconsistencies as errors and is therefore suitable for a release check. Neither command builds anything.

A typical failure makes the distinction concrete: if the `tagpax` package is missing on a new machine, doctor helps because the toolchain is incomplete. If, on the other hand, `020`-processes was changed after a successful build and the combined notes still use its old build, check reports stale project state. report is useful for investigating the cause first because it shows the builds and dependencies involved without treating the finding as a release failure.

```
ollm report --scope=series
ollm check --scope=current --target=script --language=de
ollm check --format=json
```

doctor, by contrast, checks the toolchain: Perl, `latexmk`, `LuaLaTeX`, `kpsewhich`, the TOML parser, and required TeX packages. Inside a project it also checks the manifest, definitions, writability, state, and shell-escape policy. Use doctor when the environment itself is in doubt, and check when an existing project state appears inconsistent.

```
ollm doctor
ollm doctor --project-root=/path/to/project --format=json
```

8.3 Cleaning and Maintaining State

OLLM separates build directories from document instances retained for the series. `clean --level=aux` removes auxiliary files but retains the local PDF; `build` removes the isolated build directory; `state` removes these document instances; and `all` combines the last two. Use `--dry-run` to inspect the deletion plan first. The available scopes here are `current`, `unit`, and `series`.

```
ollm clean --level=aux
ollm clean --level=build --scope=unit --dry-run
ollm clean --level=state --scope=series
```

prune removes abandoned temporary states and older, replaced build results. State belonging to physically removed units is included only with `--stale-units`. This is deliberately more cautious than deleting `.osglecture` wholesale, which also loses known mappings of logical unit IDs.

```
ollm prune --dry-run
ollm prune --stale-units
```

8.4 Deployment

deploy copies already successfully built PDFs to destinations configured in the manifest. It neither starts a build nor creates destination directories. Therefore run `build` or `build --resolve` first, then `deploy`. The default scope is `series` at the project root and `current` inside a unit.

```
ollm deploy handout --language=de --scope=series
ollm deploy --target=script --scope=collection
ollm deploy --all --overwrite
```

File-name templates can use, among other values, the series ID, unit ID, ordinal, chapter, document type, language, and role. An existing identical file remains untouched. Whether a different existing file requires `--overwrite` is controlled by `security.overwrite`.

Deployed copies can carry restricted PDF permissions, for example without permission to print. The restrictions are set in `[deployment.permissions]` as the default for all document types or in `[deployment.types.<doctype>.permissions]`; a document type's own table replaces the default completely, so `permissions = {}` lifts it. Permissions that are not mentioned remain granted.

```
[deployment.permissions]
print = "none"      # full | low | none
extract = false     # copy text and graphics
modify = false      # change page content
annotate = false    # add comments
form = false        # fill in forms and sign
assemble = false    # insert, delete, rotate pages

[deployment.types.handout]
permissions = {}     # handouts are meant to be printed
```

`--permissions` replaces the configured permissions of all document types for one run, for example `--permissions = {print=none,extract=false}`; `--permissions = {all}` deploys without restrictions.

For restricted copies, **OLLM** needs the external program `qpdf`; `ollm doctor` reports whether it is found. The copy is encrypted with AES-256 and opens without a password; the owner password is random and is not stored unless you set one yourself with `--owner-password = {PASSWORD}`. The built original remains unrestricted. Common tools remove such restrictions easily; they are a hint to readers, not access control.

If `qpdf` is missing, `deploy` asks in the terminal whether to copy the PDFs without restrictions. With `--non-interactive` or without a terminal, for example in CI, it stops with exit code 69 instead.

8.5 Creating a Project and General Options

`ollm newproject` requires the profile choice `--profiles = {taggable|classic}` and creates a new `ollmconfig.toml` and, unless it already exists, `Include/projectconfig.tex` with dummy metadata. `taggable` selects `ltx-talk` and `book` with tagging switched on and creates `Include/documentmetadata.tex`; `classic` selects `beamer` and `scrbook` without Tagged PDF. Without the option the command explains both variants and writes nothing. Single targets can be mixed by hand in the manifest afterwards. The command never overwrites an existing file. For the exact syntax of an installed version, `ollm --help` and `ollm --version` are authoritative. Common global options are `--config = {FILE}`, `--project-root = {DIRECTORY}`, `--format = {text|json}`, `--color = {auto|always|never}`, and `--non-interactive`.

For build diagnostics, `--debug = {tex}`, `ollm`, or `tex+ollm` selects the output layer; `--warnings = {all}`, `important`, or `none` controls warning verbosity. Compatible unknown dash options are forwarded to `latexmk` during a build; OLLM rejects options that would bypass the controlled build contract.

8.6 Complete Command Summary

build builds one or more instances of a series; this is the default action.

standalone builds a manifest-free source file from its class options.

report describes units, instances, versions, output files, and dependencies without validating them.

check validates the same state and returns a failing status for inconsistencies.

clean removes data selected by `--level` and `--scope`.

prune removes older, replaced build results and optionally the state of missing units.

doctor checks the toolchain and, inside a project, its basic configuration.

deploy copies already built PDFs according to the deployment rules.

convertproject converts a legacy `ollmconfig.pl` to TOML and imports recognizable metadata from `Include/lectdates.tex`.

newproject creates the manifest and TeX project configuration, using existing legacy files as conversion sources.

Targets and aliases. The built-in bare target names are `slides`, with aliases `beamer` and `presentation`, `handout`, `screen`, and `script`, with alias `article`. Select other registered targets with `--target = {TARGET}`.

All OLLM options.

- `--help, -h` show help.
- `--version` show the version.
- `--format = {text|json}` human- or machine-readable output.
- `--color = {auto|always|never}` control colored output.
- `--non-interactive` forbid interactive actions.
- `--profiles = {taggable|classic}` profile set of the generated manifest; required for `newproject`, optional for `convertproject` (default there: `classic`).
- `+enforce+`, `--enforce+` require the `+` prefix on action and target words.
- `--legacy` explicitly use the legacy `ollmconfig.pl`.
- `--overwrite` allow replacement when permitted by project policy.
- `--target = {TARGET}` select a target.
- `--language = {LANG}` select a language; compatible forms are `lang=LANG` and `+lang=LANG`.
- `--source = {FILE}` select the main source.
- `--all` select all permitted target/language combinations in the scope.
- `--resolve` rebuild dependencies to a fixed point.
- `--rebuild` force a complete `latexmk` rebuild.
- `--dry-run` show the normalized plan without executing it.
- `--level = {aux|build|state|all}` select the level for clean.
- `--scope = {current|unit|series|collection}` select the range for building, inspection, maintenance, or deployment; not every action permits every value.
- `--stale-units` include physically missing units in prune.
- `--debug = {tex|ollm|tex+ollm}` select the diagnostic layer.

--warnings = {all|important|none} select warning verbosity.

--config = {FILE} select a manifest explicitly.

--project-root = {DIR} select a project root explicitly.

Compatible unknown dash options are forwarded to `latexmk` only for builds. Options that would bypass the engine, output format, or controlled security parameters are rejected. For standalone, no project-related selection options other than **--config** and **--project-root** are permitted.

Index

Symbols

+enforce+ 55
 --all 18, 51, 55
 --config 56
 --dry-run 51 f., 55
 --enforce+ 55
 --help 55
 --language 14, 28, 51
 --legacy 55
 --level 54
 --non-interactive 54 f.
 --overwrite 15, 53, 55
 --permissions 53
 --project-root 56
 --rebuild 51, 55
 --resolve 17, 36 ff., 51, 55
 --scope 54
 --source 51
 --stale-units 53, 55
 --version 55
 -h 55

A

\ActiveLectureModes 28, 33
 adapter 50
 \AddLectureModes 48
 \anditem 34
 ansitem (package) 5
 article (class) 5
 \author 24 f., 33
 auto 49
 \autoref 35

B

babel (package) 11, 27
 backend 46, 50
 Beamer (bundle) 7
 beamer (class) 5, 12, 19, 21–24, 26, 28, 46, 54
 beamer (package) 4
 beamerswitch (package) 4
 biblatex (package) 31 f., 35, 49
 bibliography 49
 book (class) 10, 19, 22 ff., 46, 54
 bundle_preset 44
 \butitem 34

C

capabilities 50
 \cite 31, 35
 class 45 f., 50
 class-options 40, 49 f.

class/<slides> 45
 class/beamer 45
 class/NAME 45
 \colonitem 29, 34
 \columnsep 30
 \commaitem 29, 34
 comment (package) 4
 \course 25, 29, 32
 course-target 25, 50
 CTAN 4 f.

D

\date 25, 33
 \DeclareLectureModeAlias 48
 \DeclareLectureModeParents 48
 \DeclareModeBehaviorClass 48
 \DeclareOsgLectureMetadataField 45, 50
 \DeclareOsgLectureProfile 50
 \DeclareOsgLectureProfileClassOptions 50
 default_language 43
 doctype 40, 43, 48
 doctypes 43, 46, 50
 document-metadata 43, 50
 document_metadata 43, 46
 \documentclass 11, 24, 41
 \DocumentMetadata 10, 24, 38, 41 f., 46 f.

E

\EnforceLectureProjectConfiguration 45, 49
 engine 50
 \event 25, 33
 event-target 25, 50

F

filecolor 49
 \FinalizeLectureModes 48
 \finite 29, 34
 frame (environment) 12, 14, 26
 \frametitle 12, 26

H

hyperref (package) 30

I

identity-profile 49
 \IfLectureModeF 33
 \IfLectureModeT 33
 \IfLectureModeTF 26, 33
 \include 37
 \IncludeOsgLecturePreamble 45, 49
 \includeunit 17, 38, 41

INDEX

`initial` 34
`\institute` 25, 33
`\item` 22, 34

L

`lang` 31, 35, 40, 49
`langselect` (package) 4, 11, 27, 30, 43, 45, 49
`language` 10, 34
`languages` 10 f., 43, 49
`\ldeen` 12, 27
`\lecture` 11, 25 f., 31 f.
`\lecturemode` 22, 26, 28, 30, 33
`\LectureModes` 28, 33
`\LectureProjectEnforce` 49
`\LectureProjectSetup` 48 f.
`\LectureTargetSetup` 23, 49
`legacy` 50
`\litcomment` 32, 35
`\litentry` 35
`literaturelist` (environment) 35
`\litgroup` 35
`load babel` 49
`load polyglossia` 49
`\LoadClass` 45
`\longform` 34
`longform-profile` 23, 49
`longform_profile` 43, 46 f.
`\longslides` 34
`lttheme-tuc-2019` (package) 5
`ltx-talk` (class) 5, 10, 19, 21–24, 26, 28, 41, 46, 54
`ltx-talk` (package) 4 f.
`ltxtalk-theme` (package) 5

M

`map` 43, 49
`minted` (package) 5
`mode-setup-file` 48 ff.
`mode/NAME` 45
`modes` 50
`multilang` (package) 4

N

`\nameref` 35
`newpax` (package) 38
`\nextcolumn` 30, 34
`\Note` 12, 24, 26
`numbering` 49

O

`\olautoref` 16, 35
`\olnameref` 35
`\olpageref` 30, 35
`\olref` 30, 35
`options` 31

`\osgannotation` 26
`\OsgLecture` 33, 50
`osglecture` (class) 4 f., 19, 21, 23, 48
`osglecture-modes` (package) 4 f., 22
`\OsgLectureLanguage` 27, 49
`\OsgLectureReferencesSetup` 50
`\OsgLectureRequestedLanguage` 46
`\OsgLectureSet` 33, 50
`\OsgLectureSetup` 49
`\OsgLectureShort` 33, 50
`osglisings` (package) 5
`osgstyler` (package) 5
`overwrite` 43

P

`\pageref` 30, 35
`patch-overlays` 49
`patch-overlays=false` 26
`patch-overlays=true` 26
`pdfpages` (package) 38
`prefix` 49
`\prescase` 34
`presentation-profile` 23, 49
`presentation_profile` 43, 46
`\presitem` 29, 34
`presitemize` (environment) 34
`presitemize` (package) 13, 29 f.
`\preskeepcase` 34
`\prespar` 34
`\printbibliography` 31 f., 35
`\printseriesbibliography` 32, 35
`profile` 23, 40, 43, 46, 48 f.
`profile_class` 43
`project.id` 43
`punctuation` 34

R

`\ref` 16, 30, 35
`references` 49
`relation` 34
`replace` 50

S

`schema` 43
`scrarticle` (class) 5
`scrbook` (class) 14, 22 ff., 40, 46, 54
`\section` 36
`security.overwrite` 53
`selectable` 27, 49
`semcat` (package) 5
`series-target` 32
`\SetLectureModes` 48
`setup-file` 50
`shell_escape` 43

INDEX

<code>\slides</code>	34	<code>\thereforeitem</code>	34
<code>\sourceref</code>	26	<code>TikZ</code> (bundle).....	7
<code>standalone</code>	40, 48	<code>\title</code>	24 f., 33
<code>style</code>	31	<code>trim</code>	49
<code>\subtitle</code>	25, 33	<code>twocolumns</code> (environment)	30, 34
T		<code>type</code>	31, 35
<code>tagbridge</code> (package).....	5	U	
<code>tagdirtree</code> (package)	5	<code>unified shorthands</code>	49
<code>tagpax</code> (package).....	4, 38, 52	<code>unit_scopes</code>	43
TANTAU, Till.....	7	W	
<code>targetlang</code>	27, 49	<code>whitespace</code>	34
<code>tex_config</code>	43	WRIGHT, Joseph	3
<code>tex_directory</code>	43		
<code>theme</code>	49		