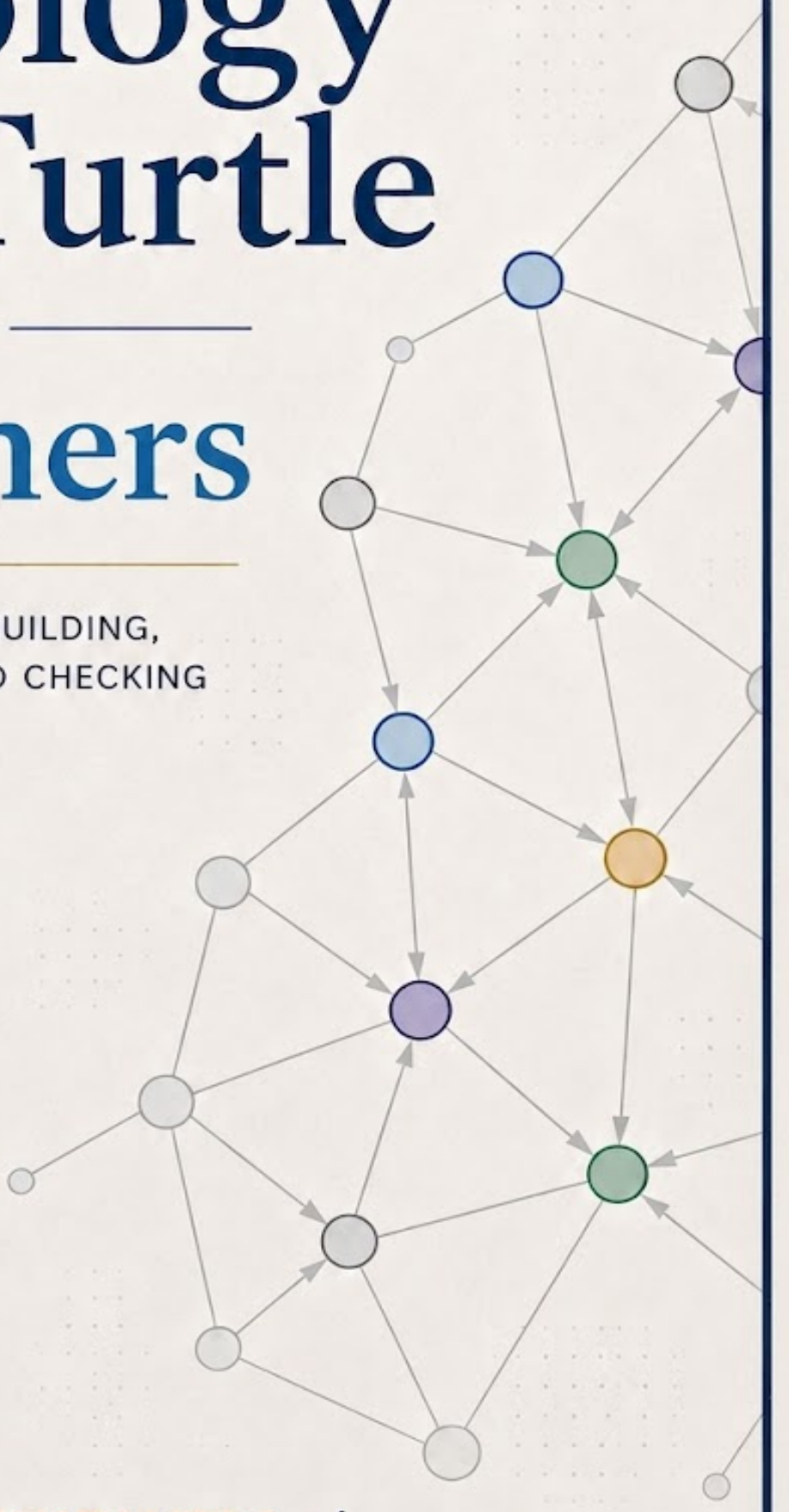
Ontology and Turtle *for* Beginners

A PRACTICAL GUIDE TO BUILDING, WRITING, QUERYING, AND CHECKING REAL-WORLD DATA

Pranav SB



SWATANTRA.ai

Ontology and Turtle for Beginners

A Complete Beginner's Guide: Introduction, Parts 1-8, and Appendices

Swatantra.ai

Combined Edition - July 2026

Contents

Ontology and Turtle for Beginners

Preface

Why This Book Exists

This book exists because of a gap I kept running into.

There is excellent material on RDF, OWL, SPARQL, SHACL, and ontology engineering. Much of it is written for readers who already know why they are reading it. A beginner is often shown formal terms or syntax before being given a concrete reason to care about them. The difficulty is not always the idea itself. Often, the difficulty is the order in which the idea arrives.

This book changes that order. You see a graph before you see a Turtle file. You learn to recognise things, relationships, values, and actions before syntax becomes the focus. You use Protégé to make modelling concepts visible before Part 3 turns those same concepts into Turtle. Later, you ask questions with SPARQL, separate asserted facts from inferred facts, and check data with SHACL. The sequence is deliberate: meaning first, notation second, automation last.

> **The aim is not to make ontology work simpler than it really is. The aim is to remove avoidable confusion so that the difficult parts are difficult for the right reasons.**

Acknowledgements

This book leans on Protégé. Parts 1 to 3 use it as the visual workbench where classes, properties, individuals, and inferred facts become something you can see rather than something you have to imagine. Protégé is developed and maintained by Stanford University and is free to download and use. Without a tool of that quality available at no cost, the visual-first approach this book depends on would not be practical for a beginner. My thanks to the team behind it.

One influence on the book's method also deserves to be named: the fact-based modelling tradition, which insists that a model should be readable back in language a domain expert can understand. Victor Morgante's work on FactEngine Knowledge Language (FEKL) uses controlled natural language to define fact-based models and business rules. The broader tradition is also visible in Object-Role Modeling and in related controlled-language work such as Clifford Heath's CQL. I found the underlying discipline valuable: a model is easier to review when a technical statement can be verbalised clearly enough for a non-specialist to challenge it.

This book is not a FactEngine, FEKL, FEQL, ORM, or CQL tutorial. Its technical exercises use RDF-family standards such as RDFS/OWL, Turtle, SPARQL, and SHACL, together with Protégé as the modelling workbench. No FactEngine product is required to complete the book. What I did take from that tradition is the habit of reading a model back in plain English before accepting it. Chapter 14 gives that habit an explicit place through ex:factReading, a custom annotation property created for this book. It stores a reusable human-readable sentence pattern beside a property. It is documentation only. It does not add inference, enforce validation, execute a query, or run a workflow.

Why the Book Separates Modelling, Asking, Checking, and Acting

A second design choice runs through the entire book: different jobs stay separate. A model describes meaning. A query asks what the graph currently says. A validation rule checks whether data is acceptable. Application code acts on a result. Beginners get into trouble when these jobs are blended and a comment is mistaken for a rule, a rule is mistaken for an inference, or a graph description is mistaken for executable workflow logic.

The examples deliberately return to this separation. The same leave-request domain may appear as a graph, a Turtle file, a SPARQL question, a SHACL shape, and finally an application boundary. The domain stays familiar so that the role of each tool becomes easier to see.

How to Read This Book

Read it with Protégé open when the chapters ask you to use it. Type the small examples rather than only reading them. Do the checkpoints. When a model looks obvious, explain it in plain English anyway. When a reasoner adds a fact, identify what was asserted and what was inferred. When SHACL rejects data, read the report before changing the file.

Most importantly, do not try to prove that you can reproduce Turtle from memory. Real ontology work is not a syntax recital. The useful skill is being able to read a model, adapt it, explain it, query it, validate it, and know which tool is doing which job.

If this book succeeds, you should finish it with more than a collection of syntax patterns. You should be able to look at a business statement, decide what belongs in the graph, express it clearly, check it, and explain the result to another person without hiding behind notation.

Pranav SB
Bengaluru, 2026

Ontology and Turtle for Beginners

Book Introduction and Start Guide

Book Introduction

Welcome. This book is for a beginner who wants to understand ontology and Turtle without being pushed into expert theory too early. You do not need to know RDF, OWL, SPARQL, SHACL, databases, programming, or formal logic before starting.

The goal is practical. You will learn to see information as connected things, relationships, and values. Then you will learn how that connected knowledge is written in Turtle, checked, queried, and explained.

By the end, you should be able to read TTL files comfortably, identify their main components, assemble and adapt small TTL packages using guided patterns, model hierarchy and workflow structures in TTL, ask useful graph questions, run provided validation code, fix failed data, and explain the model in plain English.

Important learning rule: this book does not test whether you can build TTL from memory. Real modelling is not memory performance. The skill is reading, adapting, assembling, validating, and explaining TTL safely.

You will not build a full HRMS system. The final project is intentionally small: a leave-request knowledge-graph package with a model file, example facts, hierarchy and workflow structure, questions, validation checks, a README, and an AI review log.

The teaching path is simple: see the graph first, use Protégé to understand concepts visually, then learn Turtle as the written version of what you already saw.

What the Reader Will Be Able To Do

Ability	What it means in plain English
Read TTL comfortably	Look at a TTL file and recognise prefixes, subjects, predicates, objects, classes, individuals, object properties, data properties, labels, comments, and final dots.
Build TTL with guidance	Use a clear pattern to assemble and adapt small TTL files instead of trying to reproduce syntax from memory.
Model hierarchy in TTL	Represent category or organisation hierarchy using class/subclass, part-of, or reporting-style relationships when the book teaches them.
Model workflow structure in TTL	Represent workflow steps, statuses, roles, transitions, and evidence as graph structure. TTL describes the workflow; it does not execute the workflow.
Ask and check	Use graph questions and later validation examples to find facts, expose missing data, and correct mistakes.
Explain the model	Translate the TTL back into plain English so another person can understand what the file says and why it was modelled that way.

How This Book Works

Rule	What it means
Visual first	You see relationships as simple graphs before you write syntax.
Protégé first	Protégé is used as the first visual workbench for seeing classes, properties, individuals, and relationships.
Turtle later	Turtle is taught after the visual ideas are clear, so the text format feels like a mirror of the model, not a new mystery.
Guided assembly	The reader uses patterns, examples, corrections, and checks. The book does not demand blind recall.
Small examples	The book uses small domains such as family, e-commerce, company structure, library, flight, HRMS, traffic, banking, and supply chain. Each domain teaches one idea.
Verification over trust	AI can suggest wording, spot risks, or explain errors, but the reader must verify before accepting anything.

Minimal Tool Setup Before Part 1

Only set up Protégé before starting. There is no separate setup chapter. Download and open Protégé Desktop so the reader is ready for the visual modelling sections later.

Official download page: https://protege.stanford.edu/software/

What the reader must do now:

- Go to the official Protégé software page.
- Download Protégé Desktop for the operating system being used.
- Open the application once and confirm it starts.
- Do not install extra plugins, graph databases, SHACL tools, or developer environments at the beginning.

Boundary: Protégé helps the reader see the ontology. It is not introduced here as the final proof that data is valid.

SHACL Validation Boundary

Do not set up SHACL tools before Part 1. When the book reaches validation, it will provide a small Python validation script. The reader will run the code, read the result, and fix the data. The opening section should only say that validation comes later.

Plain promise for later: SHACL checks whether the data is acceptable. The Python code is only the way the reader runs that check. The code is not the lesson by itself.

The Safe AI Rule

AI can suggest. You verify. This book does not teach the reader to outsource judgement to AI. AI may help explain a term, review a small model, or suggest a possible mistake. The reader accepts nothing until it is checked against the chapter task, Protégé view, Turtle syntax, query result, validation report, or checklist.

AI behavior	Reader behavior

AI suggests a classification.	Check whether the item is a thing, relationship, value, or action.
AI suggests Turtle.	Check the syntax and whether the statement matches the intended graph.
AI explains a validation failure.	Check the failure report before changing the file.
AI proposes extra features.	Reject scope expansion unless the chapter asks for it.

What This Book Is Not

- It is not a full enterprise architecture book.
- It is not a full HRMS product design manual.
- It is not a programming course.
- It is not a glossary disguised as a book.
- It is not a memory test for Turtle syntax.
- It is not a promise that AI output is correct.
- It is not a reason to turn every table cell, action, or number into a graph node.
- It is not a workflow engine. TTL can describe workflow structure; application logic executes the workflow.

One-Page Domain Map

The book changes examples on purpose. Each domain is a small teaching surface, not a new project.

Domain	Why it appears	What the reader should notice
Family	Simple relationships	A person can be connected to another person.
E-commerce	Categories and hierarchy	A product can belong to a category.
Company / HRMS	Business records and approvals	Employees, managers, requests, statuses, hierarchy, and workflow steps need clean links.
Library	Stable identity	A title is not the same as a specific book copy.
Flight	Events and schedules	Some facts change over time.
Traffic	Rules and context	A statement may depend on the situation.
Banking	Validation risk	Missing or wrong data can create real consequences.
Supply chain	Traceability	One thing may depend on earlier things.

Book Route

Stage	Reader learns to	Main proof
Before Part 1	Open Protégé and understand the learning promise.	Protégé opens successfully.

Part 1	Think in graph terms before code.	Classify things, relationships, values, and actions.
Part 2	See modelling concepts visually in Protégé.	Recognise classes, properties, and individuals on screen.
Part 3	Read and write clean Turtle with guidance.	Assemble and adapt small valid .ttl files.
Hierarchy modelling	Represent simple category and organisation hierarchy in TTL.	Use hierarchy relationships clearly and explain them.
Workflow modelling	Represent workflow structure in TTL without pretending TTL executes actions.	Model steps, roles, statuses, transitions, and evidence.
Reasoning section	Separate asserted facts from inferred facts.	Explain what the model lets a reasoner conclude.
SPARQL section	Ask graph questions.	Run simple graph queries.
SHACL section	Check data quality.	Use the provided Python validation code and fix a failed record.
Capstone	Assemble a small HRMS leave-request package.	Assemble, query, validate, fix, explain, and review the package.

Before You Enter Part 1

Quick start checklist:

- I know this is a beginner book, not an expert ontology manual.
- I have downloaded and opened Protégé Desktop from the official Protégé software page.
- I understand that Turtle writing comes later.
- I understand that I will assemble and adapt TTL using guided examples, not build from memory.
- I understand that SHACL validation will use a small Python code example later, not now.
- I understand that AI is a helper, not an authority.

Next: Begin Part 1 - See the Graph Before the Code.

Part 1

See the Graph Before the Code

You are entering Part 1 · See the Graph Before the Code

In this part, you will see the whole destination in miniature, learn to tell RDF and Turtle apart before they can be confused, discover why an ordinary table breaks once questions become connected, and learn to sort any business sentence into things, relationships, values, and actions to leave alone for now. By the end of Chapter 4, you will think in graph terms before you have written a single line of Turtle.

Part 1 · See the Graph Before the Code

Chapter 1

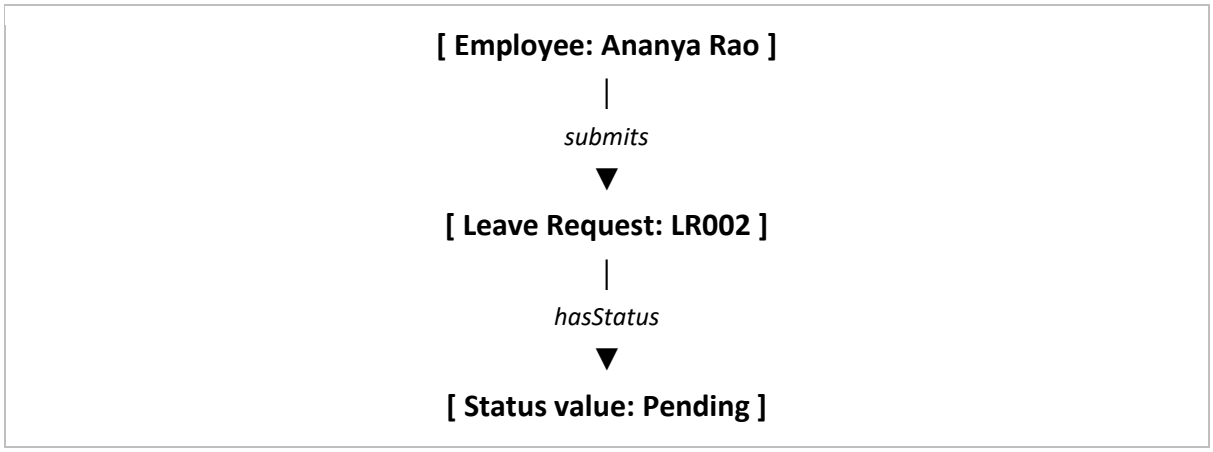
What Are We Actually Building?

Your workspace is ready. Before you learn a single technical term, you deserve to see the whole destination — the small, complete thing every later chapter is slowly building toward.

What this chapter produces: A tiny graph you can already read in plain English, the Four Jobs wall chart, and a first look at the small set of files — TBox, ABox, a SPARQL file, a SHACL file, a README, and your AI Review Log — that this book keeps returning to.

One small graph, seen whole

Every tool in this book — Protégé, Turtle, OWL, SPARQL, SHACL — exists to build, read, question, or check some version of the following small graph. This is the entire shape in miniature.



Read it top to bottom, like a short sentence: an employee submits a leave request, and that leave request currently has the status Pending. Each arrow is one visible relationship fact. Two arrows, two visible facts — and already you have a graph.

Nothing about this shape is exotic. It is the same shape as 'a parent has a child' or 'a book has a title.' The business words change from chapter to chapter — family trees, e-commerce products, flights, banks — but the underlying shape of thing → relationship → thing (or thing → relationship → value) stays close to the same. Later chapters add more patterns — classes, hierarchies, reasoning, queries, checks — but they keep returning to this basic idea: things connected to other things or values.

The four jobs you will meet again and again

Beginners usually run into trouble not because a tool is hard, but because two different jobs get blended into one. This book gives each job a one-word name now, so that later chapters can simply point back to it instead of re-explaining the confusion each time.

Job	Plain question it answers	What it is NOT

Job	Plain question it answers	What it is NOT
Model means	What kinds of things and relationships does the model name, and what do they mean?	Not a question about one specific record — it's about the vocabulary itself.
Ask asks	What does the graph currently say?	Not a rule about what should be true — just what is currently stated or inferred.
Check checks	Is this data acceptable?	Not a way to invent new facts — it only judges facts that already exist.
Code acts	Now that we know something, what should the computer do about it?	Not part of the graph itself — this is ordinary application behaviour.

📌 **Rule:** Keep these four jobs separate. A model that gives something meaning is not the same as data that has been checked, and neither one is the same as a program that acts on the result.

Each job will get its own dedicated tool later: modelling meaning belongs to OWL and RDFS (Part 4), asking belongs to SPARQL (Part 5), checking belongs to SHACL (Part 6), and acting is ordinary application logic that sits outside this book's scope. You do not need to memorise this yet — Chapter 16 brings this exact chart back with real tool names attached, and Chapter 20 gives you one more chance to sort facts by job before SPARQL begins.

The small package every chapter is building toward

This book never asks you to build something huge. Every chapter, lab, and checkpoint adds one small piece to a package that stays this size:

Piece	Plain-language job	Fully explained in
TBox	The reusable vocabulary — the kinds of things and relationships the model names and explains.	Part 3 (Chapter 15)
ABox	The actual example facts — specific employees, requests, and statuses.	Part 3 (Chapter 15)
SPARQL file	A saved question you can ask the graph, such as 'which requests are Pending?'	Part 5
SHACL file	A saved set of checks the data must pass, such as 'every request must have a status.'	Part 6
README	A short note explaining what a package contains and why.	Part 7 / Capstone
AI Review Log	A record of every AI suggestion you accepted, rejected, or changed, and why.	Created in Part 7 and used in the capstone

You are not expected to master these names now — for now, just know they are pieces of the final package.

That last row is worth pausing on: you will create AI_REVIEW_LOG.md when the project folder is introduced in Part 7. The capstone then uses that log to show which AI suggestions were accepted, changed, or rejected — with reasons.

Try it yourself before checking the answer

No graph skills are needed for this. Sort each plain-English sentence into one of the four jobs: model, ask, check, or act.

Sentence	Your answer: model / ask / check / act
"An employee can submit many leave requests."	
"How many leave requests are still Pending right now?"	
"Every leave request must have exactly one status."	
"Notify the manager by email when a request becomes Approved."	

Answer key: (1) Model — it describes a kind of relationship in general, not one record. (2) Ask — it is a question about what the graph currently says. (3) Check — it states a requirement the data must meet. (4) Act — it describes something the computer should do, not a fact about the business itself.

AI Assist

Ask AI	Using the four jobs — model, ask, check, act — sort these four plain sentences about the HRMS leave-request example. Do not write any RDF, SPARQL, or SHACL yet.
AI may suggest	A sorted list mapping each sentence to model, ask, check, or act, with a short reason for each.
You must verify	Check that 'model' answers describe a general kind of thing or relationship, 'ask' answers are questions about current data, 'check' answers state a requirement, and 'act' answers describe a computer action rather than a business fact.
Do not accept if	It writes actual SPARQL or SHACL syntax, or blends two jobs into a single answer for one sentence.

Before you leave this chapter

☑ I can describe the Ananya Rao → submits → LR002 → hasStatus → Pending graph in my own words.

☑ I can name the four jobs: model, ask, check, act.

☑ I can explain why 'asking' a question is not the same job as 'checking' data.

☑ I know that TBox, ABox, a SPARQL file, a SHACL file, a README, and the AI Review Log are the pieces this book slowly builds toward.

→ Coming up next Now that you have seen the destination, the next confusion beginners almost always hit is treating RDF and Turtle as if they were the same thing. Chapter 2 draws that line before it can cause trouble later.

Part 1 · See the Graph Before the Code

Chapter 2

RDF vs Turtle: Model vs Format

You saw the destination graph in Chapter 1. Now you will keep hearing two names — RDF and Turtle — used almost interchangeably online. They are not the same thing, and mixing them up will make every later chapter harder than it needs to be.

What this chapter produces: A permanent, plain-language separation you can state in one breath: RDF is the graph model; Turtle is one readable way to write that model down as text. No file-writing yet — that begins once Protégé has shown you the shapes in Part 2.

Two different questions

RDF and Turtle answer two different questions, and most confusion disappears once you can tell the questions apart.

Term	Plain meaning	Example
RDF	The graph model: facts stored as subject → predicate → object statements.	The Ananya Rao → submits → LR002 graph from Chapter 1.
Turtle	A compact, readable text format for writing RDF facts down.	One short line of text (shown below, to read only).
.ttl file	A saved file that holds Turtle text.	leave_request_snippet.ttl — you will start creating files like this in Part 3.

An everyday anchor: think of RDF as the logic of a dish — what ingredients relate to what steps — and Turtle as one particular way of writing that logic on a recipe card. A different recipe card format could describe the exact same dish. The dish's logic does not change just because the card's handwriting style does.

📌 **Rule:** RDF is the model. Turtle is one readable syntax for writing that model down. A .ttl file is just where that text happens to live.

A tiny Turtle line — to read, not to write yet

Here is one real Turtle line from the graph you already saw in Chapter 1. For now, only read it.

```
data:leaveRequest_LR002   ex:submittedBy   data:employee_E001 .
```

Part of the line	Plain reading

Part of the line	Plain reading
data:leaveRequest_LR002	The thing being described — the subject.
ex:submittedBy	The relationship being stated — the predicate.
data:employee_E001	The thing reached by the relationship — the object.
.	The full stop that says: this one fact is finished.

You are not writing or saving anything yet. Part 3 slows down and teaches Turtle line by line, after Protégé has already shown you the shapes visually. Between now and then, Part 2 will show you these same ideas — things, relationships, and values — visually in Protégé, without any typed syntax at all.

It is also worth knowing, briefly: Turtle is not the only way to write RDF down — formats such as RDF/XML exist too. This book only uses Turtle, but that fact alone is a useful reminder that RDF the model and Turtle the format are genuinely two different things, not two names for one thing.

Where this confusion usually comes from

Failure story: a beginner reads an online tutorial that says 'write some RDF' and assumes RDF itself is a syntax to type. What they actually see on screen is almost always Turtle. They spend a week memorising punctuation, believing they are learning 'RDF,' when they are really learning one particular way of writing RDF down. The idea (subject → predicate → object) and the notation (prefixes, dots, semicolons) are two different skills — and only one of them is Turtle.

Try it yourself before checking the answer

Mark whether each statement describes RDF, Turtle, or the file.

Statement	RDF / Turtle / file
"Facts are stored as subject-predicate-object triples."	
"The line ends with a dot."	
"The file is saved with a .ttl extension."	
"This is one of several possible ways to write the same graph as text."	

Answer key: Facts stored as triples = RDF. Line ends with a dot = Turtle. Saved with a .ttl extension = file. One of several ways to write the same graph = Turtle. Common mistake: calling Turtle 'the graph model.' Turtle is the syntax; RDF is the model that syntax represents.

AI Assist

Ask AI	Explain the difference between RDF and Turtle using an everyday analogy, without writing any Turtle syntax.
AI may suggest	An analogy such as recipe-logic vs recipe-card, or blueprint vs architectural drawing, that keeps RDF as the model and Turtle as one written format.

You must verify	Check that the analogy never calls Turtle 'the graph' or 'the data model,' and that it does not drift into teaching syntax.
Do not accept if	It claims Turtle is the data model, writes a full ontology, or introduces SPARQL, SHACL, or OWL this early.

Before you leave this chapter

☑ I can say RDF is the graph data model.

☑ I can say Turtle is one readable syntax for writing RDF as text.

☑ I can point to the subject, predicate, and object in one short Turtle line, without writing my own yet.

☑ I know a .ttl file is just where Turtle text is saved — not a different concept from RDF.

→ **Coming up next** With RDF and Turtle pulled cleanly apart, the next question isn't which language to write in — it's why you would want a graph at all instead of a familiar table. Chapter 3 shows exactly where an ordinary table starts to break.

Part 1 · See the Graph Before the Code

Chapter 3

Tables Break When Relationships Matter

You learned how normal records store facts, and you can now tell RDF and Turtle apart. Now those familiar records create a relationship problem the moment you need to understand how things connect.

What this chapter produces: A table-vs-relationship visual, one hidden relationship identified in a real HRMS row, and a clear rule for when a graph model actually earns its place.

The problem this chapter solves

Most readers start with tables because tables feel safe. A table has rows, columns, and values. That is fine when the question is simple: 'What is the leave type?' or 'How many days were requested?'

The weakness appears when the question changes from 'What is stored?' to 'How are things connected?' This book starts here because ontology thinking is relationship-first. If you cannot see the relationship problem, the tools that follow will become names to memorise instead of tools with a job to do.

A table looks complete until the question becomes connected

Look at this ordinary leave-request row. Nothing looks broken yet.

Request ID	Employee	Leave Type	Manager	Status	Days
LR001	Ananya Rao	Sick Leave	Meera Iyer	Pending	3

The row answers simple reading questions. But it hides several connections inside plain text. Ananya is not just text — she is an employee. Meera is not just text — she is a manager. Sick Leave is a leave type that may carry policy meaning later. Pending is a status that may drive a process later.

Now see the same row as a graph:

[LeaveRequest: LR001]

├– submittedBy → [Employee: Ananya Rao]

├– hasLeaveType → [LeaveType: Sick Leave]

├– hasApprover → [Manager: Meera Iyer]

└– hasStatus → [Pending]

The table did not fail because it is badly designed. It failed because the question became bigger than the row.

Sharper failure story: same name, unsafe identity

Request ID / Employee	Problem
LR004 · Ravi Kumar, Finance	Which Ravi Kumar submitted this request?
LR005 · Ravi Kumar, Sales	Which Ravi Kumar submitted this request?

The table looks fine until the system must approve the request and connect it to one real employee record. A graph mindset forces the question: is this just a display label, or does this person need a stable connection to other facts?

The hidden relationship test

📌 **Rule:** If the same value needs to connect to other facts, it is probably not just a cell value anymore.

Machine thinking: can a computer answer who needs to approve this request — or is the manager only trapped as text?

Cell text	Hidden question	Better graph view
Ananya Rao	Which employee submitted this request?	Employee → submits → LeaveRequest
Meera Iyer	Who must approve it?	LeaveRequest → hasApprover → Manager
Sick Leave	Which policy does this request follow?	LeaveRequest → hasLeaveType → LeaveType
Pending	Which requests are waiting for action?	LeaveRequest → hasStatus → Pending

A quick note on status: for now, read Pending as a simple value attached to the request — that is exactly how Chapter 4 will classify it. Later in the book, once queries and checks need to reason about a fixed set of allowed statuses, this same idea can grow into its own status node. Nothing you learn now needs to be unlearned then.

When a table is still enough

Not every column needs to become a graph. If a value never needs to connect to anything else — a free-text comment field, say, or a one-off page count — a plain column is still the right choice. This book models a relationship only when a real business question needs to cross from one thing to another. Turning every cell into a graph node on principle is not careful modelling; it is clutter, and Chapter 4 gives that trap its own name.

Try it yourself before checking the answer

Here is a new row from the same HRMS system. For each cell, decide: is it probably fine as plain text, or does it hide a relationship worth modelling?

Request ID	Employee	Leave Type	Manager	Status	Days
LR010	Meera Iyer	Casual Leave	Ravi Kumar	Approved	2

Cell	Your judgement: plain text or hidden relationship?	If hidden, what is the hidden question?
Meera Iyer (employee)		
Ravi Kumar (manager)		
Casual Leave		
Approved		
2 (days)		

Answer key: Meera Iyer, Ravi Kumar, and Casual Leave each hide a relationship — they may need to connect to an employee record, a reporting line, or a policy later. Approved hides a relationship once you need to ask which requests are in which state. 2 (days) is a plain value with nothing else to connect to yet, so it can safely stay a value — a point Chapter 4 develops fully.

AI Assist

Ask AI	Look at this leave-request table row and tell me which cells hide relationships worth modelling as a graph, and which are safe to leave as plain values.
AI may suggest	It may flag the employee name, manager name, and leave type as hiding relationships, and the day-count as a safe plain value.
You must verify	Check each suggestion against the hidden relationship test: does this value need to connect to other facts?
Do not accept if	It turns every column into a graph node, or starts writing Turtle or SPARQL before those tools have been properly introduced.

Before you leave this chapter

- ☑ I can explain why a table row can look complete while still hiding relationships.
- ☑ I can apply the hidden relationship test: does this value need to connect to other facts?
- ☑ I can give one real example of a table's plain text breaking down (the two Ravi Kumars).
- ☑ I know a table is still the right choice when nothing in it needs to connect further.

→ **Coming up next** Now that hidden relationships are visible, the next question is what to actually do with them: which words become things, which become relationships, which stay as simple values, and which to leave alone for now. Chapter 4 gives you that classification, along with the most common way beginners over-model.

Part 1 · See the Graph Before the Code

Chapter 4

Things, Relationships, Values, and What Not to Model

You learned that some table cells hide relationships. Now you need a reliable way to decide: is this a thing, a relationship, a value — or something to ignore for now?

What this chapter produces: A four-way classification — thing / relationship / value / ignore for now — applied to real HRMS words and a full sentence, plus a clear rule against over-modelling.

Four buckets, not two

Every word or phrase in a business sentence falls into one of four buckets. Most beginner mistakes come from only ever using the first two.

Bucket	Plain test	Example
Thing	Does it need its own identity, and might it connect to other facts later?	Employee, LeaveRequest, LeaveType
Relationship	Does it connect two things?	submits, hasStatus, approvedBy
Value	Is it a simple number, date, or short text attached to a thing, with no need for its own identity yet?	requestedDays = 3, startDate
Ignore for now	Is it a side effect or action the graph does not need to represent at this stage?	"send an email notification"

Classify four real words

Start with the exact words this chapter is named for.

Word or phrase	Bucket	Reason
Employee	Thing	A specific employee can have requests, a department, a manager, and history.
submits	Relationship	It connects an Employee to a LeaveRequest.
requestedDays	Value	A number attached to a request. It does not need its own identity yet.
email notification	Ignore for now	It is a side effect of a status change — the 'act' job from Chapter 1 — not a business fact the graph must store at this stage.

Classify a full sentence

Take this sentence: 'Ravi submitted a three-day casual leave request, and Nisha must review it.'

> Pause. In the sentence, circle the things, underline the relationships, and box the values, before reading the table below. If you only recognise the answer after seeing it, the skill has not formed yet.

Word or phrase	Bucket	Reason
Ravi	Thing	A specific employee can have requests, a department, a manager, and history.
submitted	Relationship	It connects Ravi to the leave request.
three-day	Value	It is a number attached to the request. It does not need its own identity yet.
casual leave	Thing	A leave type can connect to policy rules later.
leave request	Thing	The request is the central object being described.
Nisha	Thing	A manager can review many requests and report to someone else.
review	Relationship	It connects the leave request to the manager who must review it.

The hard part: values are not automatically things

Over-modelling often happens because people are scared of losing information. They turn every word into a node. That is not careful modelling — it is graph clutter.

Failure story: manager as trapped text. If 'Nisha Menon' is stored only as managerName text, the computer cannot connect her to approvals, reporting structure, workload, or review history. The name is readable, but it is not followable. When a person must connect to later facts, treat that person as a thing, not merely as text.

'3' is usually a value. It becomes a thing only if the business needs to talk about it as an object with its own identity and relationships. Plain requestedDays = 3 is just a value.

> 📌 **Rule:** Model a value as a thing only when it needs identity, reuse, relationships, governance, or explanation.

Bad model vs corrected model

Bad model	Why it is wrong	Corrected model
LeaveRequest_LR002 → requestedDays → Thing: ThreeDays	Creates a fake node for a simple number. Nothing else needs to connect to 'ThreeDays' yet.	LeaveRequest_LR002 → requestedDays → Value: 3

Bad model	Why it is wrong	Corrected model
LeaveRequest_LR002 → managerName → Value: 'Nisha Menon'	The manager is not just text if you need reporting, approval, workload, or review.	LeaveRequest_LR002 → hasApprover → Manager_Nisha

The new trap: modelling actions and side effects as things

Failure story: a beginner tries to model 'send an email to the manager' as a graph node — say, NotificationEvent — before the book has even covered events. This clutters the model with process detail that belongs to application code (the 'act' job from Chapter 1), not to the graph model, at least not this early.

📌 **Rule:** If a phrase describes an action the computer should do, it usually belongs to application logic — the 'act' job from Chapter 1 — not to the graph model, at least not this early.

Practice: classify the parts

Item	Your classification (thing / relationship / value / ignore for now)
Employee_E001	
submittedBy	
5	
Sick Leave	
Pending	
approvedBy	
send reminder email	

Answer key: Employee_E001 = thing. submittedBy = relationship. 5 = value. Sick Leave = thing. Pending = value at this stage (it can remain a value until the book later teaches controlled status vocabularies). approvedBy = relationship. send reminder email = ignore for now, since it describes an action rather than a business fact.

AI Assist

🤖 **Ask AI**	Classify these HRMS words as thing, relationship, value, or ignore for now. Explain in one sentence each. Do not write Turtle or OWL.
AI may suggest	It may classify nouns as things, verbs or relationship phrases as relationships, numbers and dates as values, and action phrases as ignore for now.
You must verify	Check whether each 'thing' really needs identity or future relationships, and whether anything marked 'ignore for now' truly describes an action rather than a hidden thing.
Do not accept if	It treats every action phrase as a thing, skips the ignore-for-now bucket entirely, or starts writing RDF/Turtle syntax.

Before you leave this chapter

☑ I can sort a word or phrase into thing, relationship, value, or ignore for now.

☑ I can explain why a value should stay a value until it needs its own identity.

☑ I can recognise when a phrase describes an action or side effect that belongs to application code, not the model, at this stage.

☑ I did not over-model: I did not turn every number or action into a graph node.

→ **Coming up next** You can now look at an ordinary business sentence and separate things from relationships from values from actions to ignore for now. That is the whole of Part 1.

Next, Part 2 opens Protégé and shows these same graph ideas visually before any Turtle writing begins.

Part 2

Protégé First: Build Concepts Visually

You are entering Part 2 · Protégé First: Build Concepts Visually

In this part, you will stop imagining graphs and start building them. Protégé becomes your workspace for five short chapters, each borrowing a different everyday domain — a family tree, an e-commerce catalog, a company org chart, a library, a flight board — so that classes, individuals, object properties, data properties, and their inverse links each arrive in a setting simple enough to see clearly. A short mini-lab names the two halves you will already have built, and Chapter 9 lets you watch a reasoner catch a real contradiction, on purpose, before it can worry you later. By the end of Chapter 9, you will have built and broken a small ontology visually, without writing a single line of Turtle.

Part 2 · Protégé First: Build Concepts Visually

Chapter 5

Family Tree: Triples Without Fear

You can already sort a business sentence into things, relationships, values, and actions to ignore for now. Chapter 5 hands you your first real tool — Protégé — and asks you to build one true triple with it, using a family tree simple enough that the tool, not the domain, gets your full attention.

What this chapter produces: One parent-child relationship built entirely inside Protégé, using the Individuals tab and one object property, hasParent. Your first preview question and preview check — plain-language habits that stand in for SPARQL and SHACL until Part 5 and Part 6 teach them properly.

Why a family tree, and why now

Every example so far — Ananya Rao submitting a leave request, Ravi's three-day casual leave — has come from the HRMS world. That was deliberate: one small business scenario, revisited, so the shape of a graph could sink in before anything else moved. This chapter deliberately changes the subject.

A family tree needs no glossary. Everyone already knows what a parent is and what a child is, so when something feels unfamiliar in this chapter, it will be Protégé asking for your attention — not the business domain. That separation matters: the rest of Part 2 keeps switching domains (products, an org chart, a library, a flight) for exactly this reason, so that each new tool idea arrives in a setting that costs you nothing to understand.

The bridge back: a parent-child link is the same shape as the Ananya Rao → submits → LeaveRequest graph from Chapter 1 — one thing, connected to another thing, by one named relationship. Only the nouns changed.

A first look at the Protégé window

Open Protégé and start a new, empty ontology (Appendix A has the full install and first-launch walkthrough — this chapter assumes Protégé is already open). Along the top you will see a row of tabs. Four of them matter for this book, and you will meet all four across Part 2:

Tab	What it holds	First used
Classes	The kinds of things your model names — the shapes, not the examples.	Chapter 6
Object Properties	Relationships that connect one thing to another thing.	Chapter 5 (this chapter)
Data Properties	Relationships that connect a thing to a plain value — text, a number, a date.	Chapter 8
Individuals	The actual examples — Priya, Kabir, one specific book, one specific flight.	Chapter 5 (this chapter)

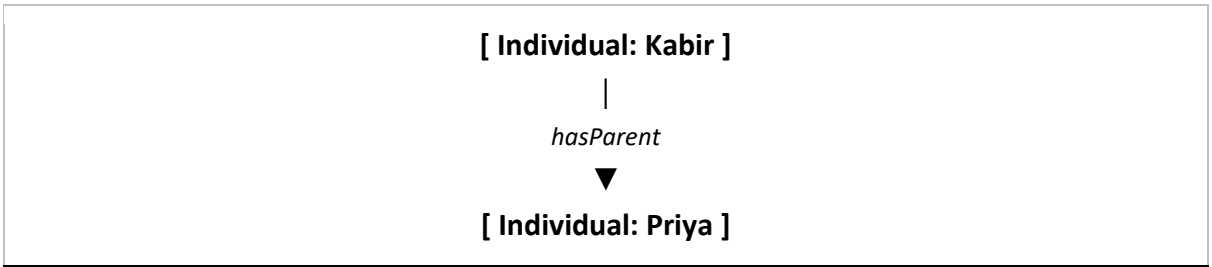
This chapter only touches two of the four: Individuals, and one Object Property. That is intentional — one new idea at a time.

Building one triple visually

The whole chapter comes down to three small steps.

Step	What you do in Protégé	What it means
1	Open the Individuals tab and create an individual named Priya.	You have named one specific thing — not a kind of thing.
2	Create a second individual named Kabir.	A second specific thing, ready to be connected.
3	Open the Object Properties tab, create a property named hasParent, then go back to Kabir in the Individuals tab and add an object property assertion: hasParent → Priya.	You have stated one relationship fact: Kabir hasParent Priya.

Read that assertion out loud once you have made it: Kabir has parent Priya. That sentence, in full, is the triple. Protégé never asks you to type a triple as one line of text — you build it from two individuals and one property link, and the tool assembles the underlying fact for you.



📌 **Rule:** In Protégé, a triple is never one line you type. It is two individuals plus one property assertion between them. The subject, predicate, and object you learned to name in Chapter 2 are still there — Protégé just builds them from clicks instead of punctuation.

Preview question and preview check

From here on, most chapters in Part 2 end their walkthrough with two small, plain-language prompts — a preview question and a preview check. Neither is real syntax yet. They exist to keep the ask and check jobs from Chapter 1's wall chart warm in your mind, long before SPARQL (Part 5) and SHACL (Part 6) give them a formal home.

Prompt	In plain English	Which Chapter 1 job it rehearses
Preview question	Who is Kabir's parent?	Ask — a question about what the graph currently says.
Preview check	Does every child individual have at least one hasParent value?	Check — a requirement the data should meet.

You do not answer these with a query language yet — you answer them by looking at the graph and reasoning it out, the same way you already read the Ananya Rao graph in Chapter 1.

Failure story: a beginner, worried about losing information, types "Priya" directly into a data property called parentName on Kabir's individual, instead of creating Priya as her own individual and linking with hasParent. The name reads correctly on screen, but the moment a later question needs Priya's own parent, or her other children, or her age, there is nothing to connect to — Priya is trapped as text, exactly the failure story Chapter 4 already warned you about.

📌 **Rule:** If a business fact needs its own identity — its own future relationships, its own attributes — model it as an individual and connect with an object property. Reach for a data property only when the fact really is a plain value with nothing more to say about it.

Practice: grow the tree

Before checking the answer, build this three-generation family in Protégé using only what this chapter taught you: individuals plus one object property, hasParent.

Individual	hasParent value	Your check: does this line follow the pattern above?
Meera	(none — Meera is the eldest generation shown here)	
Priya	Meera	
Kabir	Priya	
Anya	Priya	

Answer key: Kabir and Anya are siblings — they share the same hasParent value, Priya. Priya, in turn, has parent Meera, one generation further up. Meera has no hasParent value in this small tree — that is expected, not an error; the graph simply does not reach back any further. If you asked the preview question "Who is Anya's parent?" the graph answers it in one hop: Priya. Ask "Who is Anya's grandparent?" and the graph answers it in two hops: Priya, then Meera.

AI Assist

🤖 **Ask AI**	I have four family members and one hasParent relationship. Help me check whether my Protégé setup — individuals plus one object property — matches the triple pattern from this chapter. Do not write Turtle or any query syntax.
AI may suggest	A description of which individuals should exist and which hasParent links should connect them, stated as plain sentences ("Kabir hasParent Priya").
You must verify	That every relationship suggested is between two individuals you actually created, not a name typed as text, and that no property has been invented beyond hasParent.
Do not accept if	It suggests storing a parent's name as a data property, invents additional relationship types this chapter has not covered, or writes any Turtle or SPARQL.

Before you leave this chapter

- ☑ I can create an individual in Protégé's Individuals tab.
- ☑ I can create an object property and use it to connect two individuals.
- ☑ I can read a hasParent assertion out loud as a full sentence: X has parent Y.
- ☑ I can tell a preview question (ask) apart from a preview check (check), using the Chapter 1 wall chart.
- ☑ I did not store any person's name as a plain value when that person needed to connect to further facts.

→ **Coming up next** You just linked one specific thing to another specific thing. But a real family tree — or a real product catalog — also needs to group similar things into kinds. Chapter 6 leaves individuals aside for a moment and opens Protégé's Classes tab, using an e-commerce catalog to show what a class and a subclass actually are.

Part 2 · Protégé First: Build Concepts Visually

Chapter 6

E-Commerce Categories: Classes and Hierarchy

Priya and Kabir are two specific people, each built by hand as an individual. That works for two people. It stops working the moment you have four hundred products and need to say, in one place, what a mobile phone even is. Chapter 6 opens Protégé's Classes tab and introduces the idea of a kind of thing.

What this chapter produces: A three-level class hierarchy — Product, Electronics, MobilePhone — built in Protégé's Classes tab, with two phone individuals typed against the most specific class. Preview question and preview check only; no query syntax yet.

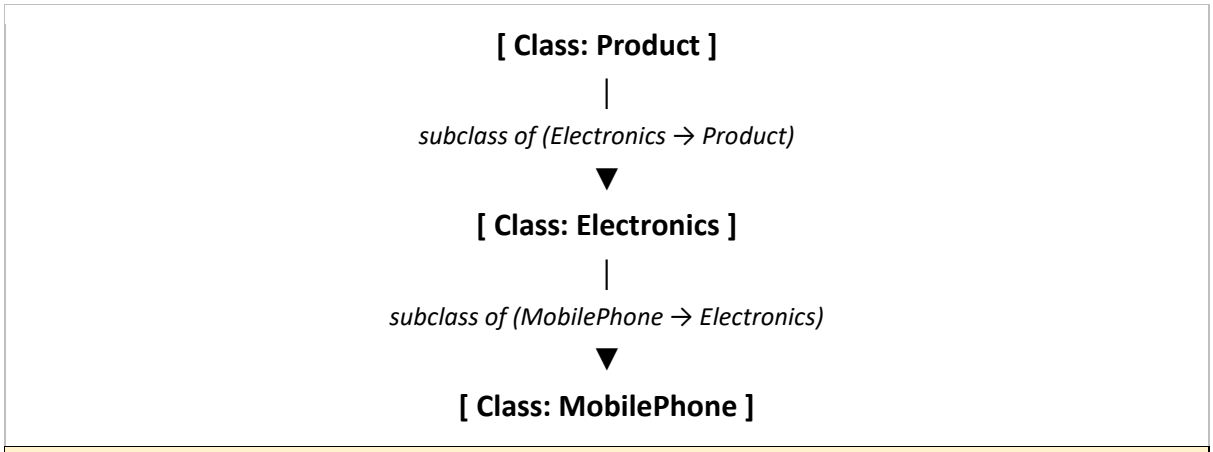
From individuals to kinds of things

Kabir and Priya were each built as one-off individuals because, in a family tree, every person genuinely is a one-off. An e-commerce catalog is different: thousands of individual products share the same kind of shape. Saying "this is a MobilePhone" once, as a class, means you never have to redescribe what a phone is for every single product row.

A class is a reusable kind. An individual is one concrete example of that kind. You already used one class informally in Chapter 1 — Employee, LeaveRequest — without naming the word "class." Chapter 6 makes that idea explicit and gives it a home in Protégé.

Building the hierarchy visually

Step	What you do in Protégé's Classes tab	What it means
1	Create a class named Product directly under owl:Thing (the root every class starts from).	Product is the broadest kind in this catalog.
2	Right-click Product and add a subclass named Electronics.	Every Electronics individual is automatically also a Product.
3	Right-click Electronics and add a subclass named MobilePhone.	Every MobilePhone individual is automatically also Electronics, and therefore also a Product.



📌 **Rule:** Subclass of means every member of the child class is automatically a member of the parent class too. Before you draw a subclass link, say the sentence out loud: "every X is a Y." If that sentence is false, the subclass line is wrong. (Part 3 will show you the written Turtle name for this same idea: rdfs:subClassOf. For Part 2, the plain phrase is enough.)

Apply the rule here: every MobilePhone is Electronics — true. Every piece of Electronics is a Product — true. The hierarchy holds because both sentences hold.

Individuals belong to the most specific class

Switch to the Individuals tab and create two individuals — iPhone15 and GalaxyS24 — both typed as MobilePhone. You do not separately mark either one as Electronics or as Product; the class hierarchy already carries that for you. Ask yourself the preview question below and you will find both individuals waiting in the answer, not calculated by hand.

You may not always see this upward membership listed as a separate, explicit assertion on screen yet — Protégé's default view does not always spell out every inferred type. For now, read it as what the hierarchy means, not as something you need to see typed out. Chapter 9 and Part 4 show how running a reasoner makes implied facts like this one visible directly in the tool.

Prompt	In plain English	Which Chapter 1 job it rehearses
Preview question	Which individuals count as Electronics?	Ask — both iPhone15 and GalaxyS24 qualify, because MobilePhone is a subclass of Electronics.
Preview check	Does every MobilePhone individual also count as a Product?	Check — true for any individual typed correctly against this hierarchy.

This quiet, automatic upward membership is your first small taste of reasoning — the topic Chapter 9 slows down and surprises you with on purpose, and Part 4 later explains in full. For now, just notice that it happens; you do not need to explain why yet.

Failure story: a beginner, unsure how much structure they are allowed to add, creates three unrelated sibling classes — Product, ElectronicsProduct, MobilePhoneProduct — with no subclass links between any of them. Every individual has to be typed against all three separately, and a later question like "show me everything electronic" has no single class to point to; the person asking it would have to know and list every leaf class by hand. A proper subclass chain avoids this entirely: ask for Electronics, and every phone, laptop, and charger already typed underneath it comes along for free.

Practice: build the hierarchy

Before checking the answer, sort these category names into a three-level subclass chain, the same shape as Product → Electronics → MobilePhone. (Product is the same top-level class you already created above; you are extending the catalog with a second branch.)

Category	Your placement (top / middle / bottom level)
Footwear	
Product	
RunningShoe	

Answer key: Product sits at the top, the broadest kind. Footwear is a subclass of Product (every piece of footwear is a kind of product in this catalog). RunningShoe is a subclass of Footwear (every running shoe is a kind of footwear). Applying the all-child-is-parent test: every RunningShoe is Footwear — true; every piece of Footwear is a Product — true. The chain holds together for the same reason the phone hierarchy did. Notice that Footwear and Electronics can happily coexist as sibling subclasses under Product; a catalog does not need every kind to fit under every other kind.

AI Assist

Ask AI	I want a three-level class hierarchy for an e-commerce catalog, similar to Product → Electronics → MobilePhone. Suggest one more branch under the same Product, and explain each subclass link using the all-child-is-parent test. Do not write Turtle or SPARQL.
AI may suggest	A short chain of two or three subclass links under Product with a one-sentence justification for each, phrased as "every X is a Y."
You must verify	That each suggested "every X is a Y" sentence is actually true for your catalog, not just plausible-sounding. Reject any link where you can think of an exception.
Do not accept if	It proposes more than three or four levels for this simple example, mixes in object or data properties, or writes any file syntax.

Before you leave this chapter

☑ I can create a class and a subclass in Protégé's Classes tab.

☑ I can apply the all-child-is-parent test before adding any subclass link.

☑ I can explain why an individual typed as the most specific class automatically counts as every class above it.

☑ I did not create sibling classes where a subclass chain belonged instead.

→ **Coming up next** You have now grouped individuals into kinds. But the object property you used in Chapter 5, hasParent, only pointed one way. Chapter 7 returns to relationships and shows you Protégé's inverse-property setting — using a small company hierarchy that will reappear, unchanged, when you reach the capstone.

Part 2 · Protégé First: Build Concepts Visually

Chapter 7

Company Hierarchy: Object Properties and Inverse Links

hasParent, from Chapter 5, only ever pointed one way: child to parent. Many real relationships are naturally two-directional — a manager manages an employee, and that same employee reports to that manager. Chapter 7 shows you how to declare that second direction once, correctly, instead of typing it by hand every time.

What this chapter produces: A small company hierarchy where manages and reportsTo are declared as inverses of each other in Protégé's Object Properties tab. Preview question and preview check only. Forward note: this exact company reappears, unchanged, in the book's capstone project.

One relationship, two directions

Consider Nisha Menon, a manager, and Ravi Kumar, one of her direct reports. There are two natural sentences here: "Nisha manages Ravi" and "Ravi reports to Nisha." These are not two separate facts about the world — they are the same fact, described from each end. A model that forces you to enter both by hand, separately, invites the two copies to quietly disagree with each other over time.

The bridge back: this is the same object-property idea from Chapter 5 — one thing connected to another thing by a named relationship. The new piece is that Protégé lets you tell it, once, that two property names describe the same link viewed from opposite ends.

Setting up inverse properties in Protégé

Step	What you do in Protégé	What it means
1	Open the Object Properties tab and create a property named manages.	manages will point from a manager individual to an employee individual.
2	Create a second property named reportsTo.	reportsTo will point the opposite way — employee to manager.
3	Select reportsTo, and in its characteristics panel set its Inverse Of field to manages (or set this on manages, pointing to reportsTo — either direction records the same link).	Protégé now treats the two properties as two views of one fact.
4	In the Individuals tab, add one assertion: Nisha_Menon manages Ravi_Kumar.	The reverse fact — Ravi_Kumar reportsTo Nisha_Menon — is now implied without being typed separately.

[Individual: Nisha Menon]

|

manages

▼

[Individual: Ravi Kumar]

(reportsTo → Nisha Menon follows automatically, as the inverse)

📌 **Rule:** Two properties can be declared inverses of each other. Assert one direction, and the reverse direction is implied — the two views stay synchronized because they are, in the end, one fact. Declaring an inverse does not, on its own, prove the relationship is correct: a wrongly-typed manager still needs a later data check (Part 6) to catch.

Preview question and preview check

Prompt	In plain English	Which Chapter 1 job it rehearses
Preview question	Who does Ravi Kumar report to?	Ask — answered by following reportsTo, even though you only typed manages.
Preview check	Does every non-CEO employee have exactly one reportsTo value?	Check — the CEO carve-out matters: Priyanka Shah, added in the practice below, has no reportsTo, and that is correct. Part 6 shows how SHACL expresses this kind of "applies to most, not all" rule cleanly.

Failure story: a beginner enters manages by hand for one pair of individuals, then later, working from a different spreadsheet, separately and manually enters reportsTo for a different pair — and gets the direction backwards for one of them. Nothing in the tool catches this, because the two properties were never told they describe the same relationship. Weeks later, a query built on reportsTo returns a manager reporting to their own direct report. Declaring the inverse relationship up front, as this chapter does, removes this particular class of contradiction by letting you assert only one direction and read either. It does not remove every kind of mistake — if you assert the wrong manager for someone, the inverse will faithfully carry that wrong fact in both directions. Correctness of the underlying assertion is a separate job, handled by SHACL and data review in Part 6.

Practice: extend the hierarchy

Before checking the answer, add one more level. Priyanka Shah is the CEO; Nisha Menon reports to her. Using only manages and reportsTo as declared inverses, fill in what you would assert in Protégé, and what follows automatically.

What you assert	What follows automatically as the inverse
Priyanka_Shah manages Nisha_Menon	
Nisha_Menon manages Ravi_Kumar	

Answer key: From Priyanka_Shah manages Nisha_Menon, the inverse Nisha_Menon reportsTo Priyanka_Shah follows automatically. From Nisha_Menon manages Ravi_Kumar, the inverse Ravi_Kumar reportsTo Nisha_Menon follows automatically. You typed two facts and received four — the other two arrived for free, and can never disagree with the ones you entered, because they are the same facts, not separate ones.

AI Assist

Ask AI	I have a manages object property and want to add its inverse, reportsTo, in Protégé. Explain in plain language what declaring the inverse does and does not change, without writing OWL or Turtle syntax.
AI may suggest	A plain-language explanation that asserting one direction automatically implies the other, keeping the two views synchronized — and a clear note that this does not, on its own, verify the underlying assertion is correct.
You must verify	That the explanation names manages and reportsTo as the same relationship viewed from two ends, and that it does not oversell inverses as a validation tool. Correctness of a specific manager-employee link is a separate concern, handled later by SHACL.
Do not accept if	It presents inverse properties as a substitute for data validation, or introduces transitive or functional property characteristics this chapter has not covered.

Before you leave this chapter

- ☑ I can create two object properties and declare one as the inverse of the other in Protégé.
- ☑ I can explain why declaring an inverse is safer than entering both directions by hand.
- ☑ I can trace a reportsTo answer even when only the manages direction was ever typed.
- ☑ I know this same company — Priyanka Shah, Nisha Menon, Ravi Kumar — returns later in the capstone.

→ **Coming up next** Object properties, so far, have always pointed from one thing to another thing. But some facts — a book's title, a phone's release year — are not about a relationship to another thing at all; they are plain values. Chapter 8 opens Protégé's Data Properties tab, using a library catalog, to show you exactly when a property should end in a value instead of an individual.

Part 2 · Protégé First: Build Concepts Visually

Chapter 8

Library Catalog: Data Properties

Every relationship you have built so far — hasParent, manages, reportsTo — connected one individual to another individual. But a book's title is not another thing; it is text. Chapter 8 opens the last of the three property tabs and teaches you when a property should end in a plain value instead.

What this chapter produces: One Book individual carrying a title, a publication year, and an ISBN, built using Protégé's Data Properties tab. Preview question and preview check only.

Object property vs data property

Chapter 4 already sorted business words into things, relationships, and values. Protégé's two property tabs mirror that split exactly:

Property type	Connects a thing to...	Example from this book
Object property	another thing (another individual, with its own identity)	Kabir hasParent Priya; Nisha manages Ravi
Data property	a plain value (text, a number, a date)	Book hasTitle "The Hobbit"; Book hasPublicationYear 1937

The test is the same one you already know from Chapter 4: does this fact need its own identity, its own future relationships? If yes, it belongs on the object-property side, as a linked individual. If it is simply a fact to record and nothing more, it belongs on the data-property side, as a value.

Building data properties visually

Step	What you do in Protégé's Data Properties tab	What it means
1	Create a data property named hasTitle, with range xsd:string.	Any value assigned through hasTitle will be read as plain text.
2	Create a data property named hasPublicationYear, with range xsd:gYear.	Values will be read as a year, not free text.
3	Create a data property named hasISBN, with range xsd:string.	An ISBN looks numeric but is treated as a text identifier, not a quantity to calculate with.
4	In the Individuals tab, create Book_TheHobbit and assign hasTitle = "The Hobbit", hasPublicationYear = 1937, hasISBN = "978-0261102217".	One individual now carries three plain-value facts.

[Individual: Book_TheHobbit]

├– hasTitle → "The Hobbit"

├– hasPublicationYear → 1937

└– hasISBN → "978-0261102217"

📌 **Rule:** Choose an object property when a fact could need its own identity or future relationships later (an Author individual who could also have a nationality, other books, an awards list). Choose a data property when the fact is genuinely just a value, with nothing more to attach to it (an ISBN string, a page count, a publication year).

Preview question and preview check

Prompt	In plain English	Which Chapter 1 job it rehearses
Preview question	What year was The Hobbit published?	Ask — read directly off the hasPublicationYear value.
Preview check	Does every Book individual have a title?	Check — a requirement worth stating now, formalised with SHACL in Part 6.

Failure story: a beginner records a book's author as a plain data property — hasAuthor = "J.R.R. Tolkien", a text string. The catalog works fine until someone asks "which other books did this author write?" or "list authors born in the same country." The author's name is readable on screen, but it is not followable: there is no individual behind the text to attach a nationality, a birth year, or a list of other works to. This is the exact same trap Chapter 4 named for a manager stored as text — a fact that needed identity got stored as a value instead. The fix, once the catalog needs it, is to model Author as its own class and connect books to author individuals with an object property, hasAuthor, the same way Chapter 5 connected Kabir to Priya.

Practice: classify the catalog fields

Before checking the answer, sort each library fact into thing, relationship, value, or ignore for now — the same four buckets from Chapter 4.

Field	Your classification
Book_TheHobbit	
hasTitle	
1937	
hasAuthor (pointing to an Author individual)	
"send overdue notice"	

Answer key: Book_TheHobbit = thing (an individual). hasTitle = relationship — specifically a data property, connecting the book to a value. 1937 = value. hasAuthor = relationship — specifically an object property, since an author can have their own future facts attached. "Send overdue notice" = ignore for now; it describes an action for application code, not a fact about the catalog itself, echoing the action trap Chapter 4 already warned about.

AI Assist

Ask AI	For a small library catalog, which book facts should be data properties and which should be object properties? Use title, publication year, ISBN, and author. Explain each choice in one sentence. Do not write Turtle or SPARQL.
AI may suggest	Title, publication year, and ISBN as data properties; author as an object property pointing to a separate Author individual, with a reason for each.
You must verify	That each "data property" answer is genuinely a plain value with nothing more to attach to it, and that author is flagged as needing its own identity rather than staying as text.
Do not accept if	It stores author as a data property without comment, or introduces classes and subclass hierarchies this chapter has not covered.

Before you leave this chapter

☑ I can create a data property and set its range (xsd:string, xsd:gYear, and so on).

☑ I can assign a data property value to an individual in the Individuals tab.

☑ I can apply the same thing-needs-identity test from Chapter 4 to choose between an object property and a data property.

☑ I did not store a fact as plain text when it needed to connect to other individuals later.

→ **Coming up next** Chapters 5 through 8 built classes, individuals, object properties, and data properties — one at a time, in isolation. Before Chapter 9 introduces its first real surprise, a short visual pause names the two halves you have already been building without knowing their names.

Part 2 · Protégé First: Build Concepts Visually

Mini-Lab (after Chapter 8)

See TBox and ABox in Protégé

This is a short visual pause, not a new numbered chapter. Chapters 5 through 8 already built both halves of every ontology you will ever make — you just were not told their names yet.

What this mini-lab produces: A one-page mental map: which Protégé tabs you have already been using make up the TBox, and which make up the ABox. Part 3, Chapter 15, returns to this exact split once you are ready to write it as files.

Two halves, already built

Look back across Chapters 5 to 8. In every one, you did two different kinds of work without separating them in your mind: sometimes you defined a kind of thing or a kind of relationship — reusable, one-time work — and sometimes you recorded one specific example — Kabir, Nisha Menon, The Hobbit. Those two kinds of work have names.

Protégé tab	What you built there	Reusable model term (TBox) or example fact (ABox)?
Classes	Product, Electronics, MobilePhone	TBox — reusable model vocabulary
Object Properties	hasParent, manages, reportsTo	TBox — reusable model vocabulary
Data Properties	hasTitle, hasPublicationYear, hasISBN	TBox — reusable model vocabulary
Individuals	Kabir, Priya, Nisha Menon, Ravi Kumar, Book_TheHobbit	ABox — specific example facts

📌 **Rule:** TBox is the vocabulary: the classes and properties that stay the same across every example. ABox is the data: the individuals and assertions that change from one situation to the next. You have been filling both since Chapter 5 — this pause just gives each half its name.

One more piece belongs in this picture: annotations. Every label you typed for a class, property, or individual — the human-readable name shown on screen — is an annotation. Annotations make either half readable to a person; they do not change what the model means to a reasoner.

One small annotation exercise

Before leaving this pause, add one label or comment to each of the four items below, using the annotation view/panel for the selected entity. This is the smallest possible taste of annotations; Chapter 14 in Part 3 returns to them properly and names each annotation property in Turtle.

Item you already created	Protégé annotation to add
Product (class)	Add rdfs:label: "Product"
hasParent (object property)	Add rdfs:label: "has parent"
Book_TheHobbit (individual)	Add rdfs:label: "The Hobbit"
MobilePhone (class)	Add rdfs:comment: "A product category for mobile phones."

Notice that adding these labels changes what you read on screen but does not change the class hierarchy, the individuals, or any relationship between them. That is the defining feature of annotations: they help humans, not reasoners.

Before you leave this mini-lab

☑ I can point to something I built in Chapters 5-8 and say whether it belongs to the TBox or the ABox.

☑ I added at least one rdfs:label and one rdfs:comment in Protégé's annotation panel.

☑ I know that annotations (labels and comments) make a model readable, but are not themselves TBox or ABox facts.

☑ I know Chapter 15, in Part 3, returns to the TBox/ABox split once files — not just Protégé tabs — are involved.

→ **Coming up next** Everything so far has behaved exactly as expected. Chapter 9 changes that on purpose, using a flight-status example to show you what happens when a model contains a hidden contradiction — and why that surprise is the model doing its job correctly, not a tool malfunctioning.

Part 2 · Protégé First: Build Concepts Visually

Chapter 9

Flight Status: A Model That Doesn't Add Up

Every example so far has behaved exactly as built. Chapter 9 breaks that pattern on purpose. You will build a small, ordinary-looking flight-status model, watch Protégé's reasoner catch a contradiction inside it, and learn to read that moment as good news, not a malfunction.

What this chapter produces: Two classes, OnTimeFlight and DelayedFlight, marked disjoint; one flight individual accidentally typed as both; and a first run of Protégé's reasoner, which flags the contradiction. An explicit callback to the Chapter 1 wall chart — this is the model means job catching a problem, not the check job.

Setting up two classes that cannot overlap

Create a class Flight, and give it two subclasses: OnTimeFlight and DelayedFlight. In plain business terms, a single flight should never be both at once — a flight that lands on schedule is not, at the same time, a delayed flight. Protégé lets you state that exclusion directly, rather than hoping no one ever types it wrong.

Step	What you do in Protégé	What it means
1	Create class Flight, with subclasses OnTimeFlight and DelayedFlight.	Two specific kinds of flight status, both narrower than Flight.
2	Select OnTimeFlight, right-click, and choose Disjoint With, then select DelayedFlight.	You have told the model: no individual may belong to both classes at once.

📌 **Rule:** Disjoint classes tell the reasoner that two kinds of things share no members at all. This is a stronger statement than simply not linking the classes — it is an explicit promise that overlap is impossible.

Teaching-example note: OnTimeFlight and DelayedFlight are modelled here as classes to make the disjointness surprise easy to see. A real flight-status system would usually model status as a changing value on a flight (or as a history of status events over time), because a single flight can be on time in the morning and delayed by the afternoon. This chapter is deliberately not that system — it uses classes because they make the impossibility concrete. Later parts of this book will show status as a value where that fits better.

Creating the contradiction

Now create one individual, Flight_AI202. Suppose two different systems both write status updates for this flight: an on-time feed types it as OnTimeFlight, and, minutes later, a delay feed — unaware of the first update — also types it as DelayedFlight. In Protégé's Individuals tab, assign Flight_AI202 both types.

[Individual: Flight_AI202]

types: OnTimeFlight AND DelayedFlight

✕ Disjoint classes cannot share a member

Nothing looks obviously wrong on screen yet — both type assertions are individually ordinary. The contradiction only becomes visible once you ask the model to think about what it means.

Running the reasoner

Open the Reasoner menu, choose an available reasoner such as HermiT if it is present, and start it. Appendix A shows the exact setup path for the version used in this book. Whichever reasoner you use, it finishes almost immediately for a model this small. When it does, Flight_AI202 — or the ontology as a whole — is flagged as inconsistent, and Protégé offers an Explain option that walks back through the exact logical steps that produced the contradiction: Flight_AI202 is OnTimeFlight; Flight_AI202 is DelayedFlight; OnTimeFlight and DelayedFlight are disjoint; therefore no valid model exists in which Flight_AI202 has both types.

Once the reasoner has been started, it must be stopped again before you edit the ontology further — otherwise it tries to re-run on every change. This is a housekeeping detail, not a conceptual one; Appendix A covers the mechanics in full.

Why this is good news, tied back to Chapter 1

Recall the four jobs from Chapter 1: model means, ask asks, check checks, code acts. This inconsistency is the model job working correctly. You told the reasoner, up front, what OnTimeFlight and DelayedFlight mean and how they relate — and it is now holding your data to that meaning. This is not the same thing as a SHACL check failing, which you will meet properly in Part 6: a SHACL failure says this data is unacceptable under our rules, while an OWL inconsistency says no possible situation could make this data true at all. The first is a business judgement; the second is a logical impossibility. Chapter 16 returns to this distinction with the full wall chart and real tool names attached.

📌 **Rule:** An inconsistent ontology is not a bug in Protégé. It means the facts you asserted cannot all be true at the same time, given the meaning you declared. The fix is always to correct either the facts or the meaning — never to distrust the tool for reporting it.

Failure story: a beginner runs the reasoner, sees the word "inconsistent," assumes Protégé itself is broken, and reinstalls the application. The reinstall changes nothing, because the contradiction lives in the ontology file, not in the software. The real fix is to look at the Explain output, see that one flight was typed as both OnTimeFlight and DelayedFlight, and correct the data — in this case, deciding which status feed should have won, or introducing a later status that supersedes the earlier one.

Practice: predict before you run the reasoner

Before starting the reasoner, decide which of these individuals will be flagged as inconsistent, given that OnTimeFlight and DelayedFlight are disjoint.

Individual	Types asserted	Your prediction: consistent or inconsistent?
Flight_BA118	OnTimeFlight only	
Flight_QF9	DelayedFlight only	
Flight_EK500	OnTimeFlight and DelayedFlight	

Answer key: Flight_BA118 is consistent — one type only, no conflict. Flight_QF9 is consistent for the same reason. Flight_EK500 is inconsistent — it repeats the exact contradiction this chapter built, holding both disjoint types at once. Notice that consistency is a property of what was asserted, not of how careful the modeller felt at the time; the reasoner does not guess intent, it only checks whether all the asserted facts could be true together.

AI Assist

Ask AI	I declared OnTimeFlight and DelayedFlight as disjoint classes, then accidentally typed one flight individual as both. Explain why the reasoner calls this inconsistent, using the Chapter 1 four-jobs wall chart, and without proposing a SHACL rule.
AI may suggest	An explanation that disjoint classes forbid shared membership, and that the reasoner is enforcing meaning you already declared, not inventing a new rule.
You must verify	That the explanation clearly separates this from a SHACL/data-acceptability failure, and that it points to the specific asserted facts causing the conflict rather than a vague description of "an error."
Do not accept if	It suggests removing the disjoint declaration just to make the error disappear, or introduces SHACL syntax before Part 6.

Before you leave this chapter

☑ I can mark two classes as disjoint in Protégé.

☑ I can run the reasoner (Reasoner menu → Start Reasoner) and locate the Explain option for an inconsistency.

☑ I can explain, in my own words, why an OWL inconsistency is a model-meaning problem, not a data-acceptability problem.

☑ I did not treat an inconsistency report as a software bug to work around.

→ Coming up next Part 1 built the habit of seeing a graph. Part 2 gave that habit a tool: classes, individuals, object properties, data properties, inverse links, and one honest look at what happens when a model contradicts itself. Nothing in Part 2 required you to type a single line of Turtle. Part 3 changes that — it takes everything you just built visually and shows you its written mirror, one line at a time.

Part 3

Turtle Becomes the Written Form

You are entering Part 3 · Turtle Becomes the Written Form

In this part, you stop imagining Turtle and stop watching Protégé build it for you — you start reading, writing, and splitting it yourself. Chapter 10 shows you a complete tiny file before asking you to write a single character. Chapter 11 explains why every identifier in that file looks like a machine code instead of a person's name. Chapter 12 teaches the small set of punctuation marks that hold a file together. Chapter 13 walks back through every domain from Part 2 and shows you its Turtle mirror, line by line. Chapter 14 gives your labels and comments a written form, and draws a hard line between a note for a human and a rule for a machine. Chapter 15 splits reusable vocabulary from specific facts into two real files. By the end of Checkpoint A, you will have assembled, adapted, and explained a small, correct TTL package — with no memorisation required.

Part 3 · Turtle Becomes the Written Form

Chapter 10

Read a Tiny Turtle File Before Writing One

Part 2 built five small ontologies by hand in Protégé, using only clicks — you never typed a single character of Turtle. Chapter 10 shows you, for the first time, what those same clicks look like once someone writes them down as text, before asking you to write any yourself.

What this chapter produces: one complete, twelve-line Turtle file, mostly built from facts you already understand from Protégé, explained one piece at a time — plus one small extension (an approver line) that reuses a pattern you have already practised, rather than introducing a new idea.

The file you are about to read

Recall Ananya Rao's leave request from Chapter 1 — data:employee_E001 submits data:leaveRequest_LR002, which currently hasStatus “Pending.” Below is that exact fact, plus one small addition (her manager, as approver), written as one small Turtle file. Read it once, slowly, before the line-by-line walkthrough below.

```
@prefix ex:   <https://example.org/hrms/ontology/> .
@prefix data: <https://example.org/hrms/data/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

data:employee_E001 rdfs:label "Ananya Rao" .

data:leaveRequest_LR002
    ex:submittedBy   data:employee_E001 ;
    ex:hasLeaveType  data:leaveType_CasualLeave ;
    ex:requestedDays 3 ;
    ex:hasApprover   data:employee_E003 ;
    ex:hasStatus     "Pending" .
```

⚠ **Note:** This first file is a reading snippet, not the full ontology package. It shows triples you already understand, but it does not yet declare which of ex:submittedBy, ex:hasLeaveType, ex:requestedDays, ex:hasApprover, and ex:hasStatus are object properties versus data properties, or which classes exist. Opening a file like this in Protégé now would show the individuals and their assertions, but the Classes and Properties tabs would still look empty. Chapter 15 completes the split into a full TBox and ABox.

The same file, read back to what you already built

Line(s)	What it says	Where you already built this in Protégé
Lines 1–3 (@prefix)	A shorthand setup so the rest of the file can use short names instead of long web addresses.	You never saw this screen directly — Protégé manages this behind the scenes when you create an ontology. Chapter 11 explains why it exists.
Line 5 (rdfs:label)	“Ananya Rao” is a human-readable name attached to data:employee_E001.	The Mini-Lab after Chapter 8, where you added an rdfs:label to an individual in the annotation panel.
Line 7 (subject)	Everything below this line describes one thing: data:leaveRequest_LR002.	The Individual you created for Ananya's leave request.
Line 8 (ex:submittedBy)	One relationship fact: this request was submitted by data:employee_E001.	An object property assertion, exactly like Kabir hasParent Priya in Chapter 5 — just written as text instead of clicked.
Line 9 (ex:hasLeaveType)	The request points to a leave type individual, not a plain word.	Chapter 4's classification exercise, where “casual leave” was marked a thing, not a value, because it needed to stay connected to policy rules later.
Line 10 (ex:requestedDays)	A plain number, 3, with nothing more attached to it.	Chapter 4's rule: a value stays a value until it needs its own identity.
Line 11 (ex:hasApprover)	A second relationship fact, pointing to the manager who must review this request.	Chapter 7's manages/reportsTo pair — the same kind of link, applied here to approval instead of reporting.
Line 12 (ex:hasStatus)	A plain text value, “Pending.”	The Chapter 1 graph you first saw: LR002 hasStatus Pending.

📌 **Rule:** Almost everything in this file is a fact you already built in Part 2 — every individual and most property assertions already exist as something you clicked into being. The one addition, hasApprover, is not a new kind of idea either: it reuses the exact object-property pattern from Chapter 7, just pointed at a different relationship. Turtle does not add meaning on its own — it only writes existing meaning down as portable text.

Failure story: a beginner opens a real, public .ttl file for the first time, sees forty dense lines of punctuation, and assumes ontology work requires memorising syntax before understanding anything. They close the file and give up. Look again at the file above: read patiently, it is the same handful of patterns repeated — a subject, a few semicolon-joined facts, a closing dot. Density is not the same thing as difficulty.

Try it yourself before checking the answer

Match each line group from the file above to the Protégé tab or action that produced it. Use: Individuals tab, Object Properties tab, Annotation panel, or “none — setup only.”

Line group	Your answer
Lines 1–3 (@prefix)	
Line 5 (rdfs:label "Ananya Rao")	
Line 8 (ex:submittedBy)	
Line 11 (ex:hasApprover)	

Answer key: Lines 1–3 = none, setup only — Protégé handles this without showing you the line. Line 5 = Annotation panel, the same rdfs:label habit from the Mini-Lab. Line 8 = Object Properties tab, the same assertion pattern as Kabir hasParent Priya. Line 11 = Object Properties tab again — a second, different object property on the same subject.

AI Assist

Ask AI	Here is a small Turtle file. Explain each line in plain English, tying it back to Protégé's Classes, Individuals, and Properties tabs. Do not introduce SPARQL, SHACL, or OWL restrictions yet.
AI may suggest	A line-by-line plain-English description, naming which Protégé tab or panel each idea maps to.
You must verify	That the explanation treats ex:hasApprover as a small extension using a familiar object-property pattern, not as a brand-new modelling concept — and that it doesn't drift into syntax rules Chapter 12 hasn't taught yet.
Do not accept if	It introduces classes, properties, or reasoning explanations that go beyond this one small file, or starts teaching punctuation rules ahead of Chapter 12.

Before you leave this chapter

☑ I can read a short Turtle file and point to its prefixes, subjects, predicates, and objects.

☑ I can connect every line in this file back to something I built in Protégé in Part 2.

☑ I know a Turtle file adds no new meaning — it only writes down meaning that already exists.

☑ I did not try to memorise this file — I read it for meaning, the same way I read the Chapter 1 graph.

→ **Coming up next** Every identifier in this file — data:employee_E001, data:leaveRequest_LR002 — looks like a machine label instead of a person's name. Chapter 11 explains exactly why, and why “Ravi Kumar” would make a poor identifier even though it reads perfectly well to a human.

Part 3 · Turtle Becomes the Written Form

Chapter 11

Identity and IRIs

You just read a whole Turtle file without writing one. You may have noticed something odd: every subject was a code like data:employee_E001, never a plain name like "Ananya Rao." Chapter 11 explains why that is not a stylistic habit — it is the difference between an identity a machine can trust and a label that can quietly change underneath it.

What this chapter produces: a clear rule for choosing identifiers, one safe/unsafe identifier table, and three pieces of vocabulary: IRI, prefix, and the difference between an identifier and a label.

What Protégé already showed you, without naming it

In Part 2's Company Hierarchy chapter, you typed individuals directly as Nisha_Menon, Ravi_Kumar, and Priyanka_Shah. That was a reasonable teaching shortcut — every name in that small class was unique, so nothing broke. Chapter 11 asks a harder question: what happens once you are not the only person adding data, and two employees happen to share a name?

An IRI is a stable address, not a description

Every "thing" in RDF is named by a long, globally unique, web-style address called an IRI — for example, https://example.org/hrms/data/employee_E001. Writing that out in full every time would make any file unreadable, so Turtle lets you declare a short prefix that stands in for the long part. Once declared, data:employee_E001 means exactly the same thing as the full address — it is shorthand, not a different identity.

Full IRI	Prefix declared	Short form used in this book
https://example.org/hrms/ontology/submittedBy	ex:	ex:submittedBy
https://example.org/hrms/data/employee_E001	data:	data:employee_E001

📌 **Rule:** An IRI's job is to point at the same thing, reliably, forever. It is not required to be readable, memorable, or meaningful to a human — that job belongs to rdfs:label, which you already used in the Part 2 Mini-Lab.

Why a person's name is a poor identifier

Failure story: an HRMS is built using an employee's name as their identity. Two employees are both named Ravi Kumar — a common name — and a leave request meant for one gets linked to the other, because nothing in the data distinguishes them. A second version of the same failure: Nisha Menon later changes her legal name. Every triple that used “Nisha Menon” as an identifier is now silently wrong, because identity was built from something that was always free to change.

Unsafe as an identifier	Why	Safe alternative
"Ravi Kumar"	Not unique — a common name, and one a person can legally change.	data:employee_E002
"leave request 002"	Contains a space; reads like a label, not a fixed code.	data:leaveRequest_LR002
"Manager Nisha"	Mixes a role with a name; neither part is stable on its own.	data:employee_E003

📌 **Rule:** Model an identifier for stability, and a label for readability. One record can carry both — data:employee_E002 as the identity, rdfs:label "Ravi Kumar" as the readable name — without conflict, because they do two different jobs.

The only IRI you will ever type in full

In this book, you will almost always use the short prefixed form — ex: or data: — and never type the full https:// address by hand, except once, inside the @prefix line itself. Chapter 12 shows exactly how that line is written, and what the rest of its punctuation means.

Try it yourself before checking the answer

Mark each candidate identifier as safe or unsafe, and say why in one short phrase.

Candidate identifier	Safe or unsafe?	Why
"Priya"		
data:employee_E010		
"the CEO"		
data:leaveType_CasualLeave		

Answer key: “Priya” is unsafe — a bare human name, not guaranteed unique, and one that can change. data:employee_E010 is safe — a stable code, independent of any person's current name. “The CEO” is unsafe — a role that different people hold at different times, not one fixed thing. data:leaveType_CasualLeave is safe — a stable code naming one specific leave type, regardless of who requests it.

AI Assist

Ask AI	I'm choosing identifiers for employees and leave requests in my ontology. Review this list and tell me which are safe as IRIs and which are not, and why.
AI may suggest	Flagging names, roles, and other changeable values as unsafe, and recommending stable, code-style identifiers instead.
You must verify	That every “safe” suggestion doesn't secretly still embed changeable information — for example, an identifier built from a job title or a name that depends on marital status.
Do not accept if	It suggests using full names as identifiers “because it's more readable,” or starts writing full Turtle syntax before Chapter 12 has taught it.

Before you leave this chapter

☑ I can say what an IRI is: a stable, globally unique address for one specific thing.

☑ I can tell an identifier, for machines, apart from a label, for people.

☑ I can explain why a person's name, a role, or free text makes a poor identifier.

☑ I did not choose an identifier that secretly depends on something that could change.

→ **Coming up next** Now that you can choose a safe identifier, Chapter 12 teaches the punctuation that holds a Turtle file together — the dots, semicolons, and commas that decide where one fact ends and the next begins.

Part 3 · Turtle Becomes the Written Form

Chapter 12

Turtle Syntax

You can now choose a safe identifier and read a short Turtle file for meaning. The one thing you have not yet learned is the punctuation itself — why some lines end in a dot, some in a semicolon, and some in a comma. Get this wrong, and a perfectly good model refuses to load, for reasons that have nothing to do with what you were trying to say.

What this chapter produces: working knowledge of five punctuation marks — the prefix line, the dot, the semicolon, the comma, and the quoted or typed literal — and one Turtle block you fix yourself, using only the patterns in this chapter.

Five small marks, one job each

Mark	Job	Example
@prefix	Declares a shorthand for a long IRI.	@prefix data: <https://example.org/hrms/data/> .
. (dot)	Ends a prefix line, or closes out everything said about one subject.	data:leaveRequest_LR002 ex:hasStatus "Pending" .
; (semicolon)	Keeps the same subject, adds another predicate.	ex:submittedBy data:employee_E001 ; ex:hasStatus "Pending" .
, (comma)	Keeps the same subject and predicate, adds another object.	ex:visibleTo data:employee_E003, data:employee_E004 .
"..." / "..."^^xsd:type	Marks a literal value, optionally tagged with its datatype.	"Pending" / "2026-06-20"^^xsd:date

Reading the Chapter 10 file with formal names attached

Look back at the file from Chapter 10. The subject data:leaveRequest_LR002 appears once. Every semicolon after it says “keep talking about this same subject.” The very last line — ex:hasStatus "Pending" — ends in a dot, because that is where the description of this one subject finishes.

📌 **Rule:** A dot means: I am done describing this subject. A semicolon means: same subject, one more fact. Mixing these two up is the single most common reason a Turtle file fails to load.

The tiny mistake that breaks the whole file

Bad Turtle	Why it fails	Corrected Turtle
data:leaveRequest_LR002 ex:submittedBy data:employee_E001 ;	A trailing semicolon with nothing after it — the parser expects one more fact and finds the end of the file instead.	data:leaveRequest_LR002 ex:submittedBy data:employee_E001 .
@prefix ex: <https://example.org/hrms/ontology/>	The prefix declaration never closes — it is missing its final dot.	@prefix ex: <https://example.org/hrms/ontology/> .

Failure story: a beginner writes a clean five-line block and, out of habit, ends the very last predicate with a semicolon, since every earlier line used one. The file will not load, and the error message points to a location that looks unrelated to the real mistake, because the parser only detects the problem once it runs out of file to read. The fix is always the same: check the very last predicate-object pair for a subject, and make sure it ends in a dot, not a semicolon.

Writing plain values safely

Value	Written in Turtle	Meaning
Pending	"Pending"	A plain text literal.
3	3	An integer number — no quotes, no datatype tag needed.
2026-06-20	"2026-06-20"^^xsd:date	An explicitly typed date literal.

⚠ **Note:** The exercise below does not use a typed date, but whenever you write a date literal such as "2026-06-20"^^xsd:date, you must declare the xsd: prefix: @prefix xsd: <http://www.w3.org/2001/XMLSchema#> . Any prefix used in a file — xsd: included — has to be declared before the file can load.

Try it yourself before checking the answer

Ravi Kumar has also submitted a casual leave request. Fill in each blank with the correct punctuation mark: dot, semicolon, or comma.

```
@prefix ex:   <https://example.org/hrms/ontology/> __
@prefix data: <https://example.org/hrms/data/> __

data:leaveRequest_LR003
    ex:submittedBy   data:employee_E002 __
    ex:hasLeaveType  data:leaveType_CasualLeave __
    ex:requestedDays 3 __
    ex:hasStatus     "Pending" __
```

Answer key: In order: dot, dot, semicolon, semicolon, semicolon, dot. The two @prefix lines each close with a dot because each is a complete, self-contained declaration. The first three predicate lines end in a semicolon because the same subject, data:leaveRequest_LR003, keeps being described. The final line ends in a dot because that is the last fact about this subject.

AI Assist

Ask AI	Check this Turtle snippet only for punctuation mistakes — dots, semicolons, commas, and quotes. Do not change the model or suggest new properties.
AI may suggest	Pointing out a missing final dot, a stray trailing semicolon, or an unclosed quote.
You must verify	That every suggested fix is purely punctuation — reject anything that adds, renames, or removes a property or individual.
Do not accept if	It rewrites the model, adds SHACL or SPARQL syntax, or “fixes” a line that did not actually need fixing.

Before you leave this chapter

☑ I can write a @prefix line that ends correctly with a dot.

☑ I can choose between a dot, a semicolon, and a comma when continuing a Turtle statement.

☑ I can write a plain text literal, a bare number, and a typed date literal.

☑ I can find a missing final dot by checking the very last line for a subject.

→ **Coming up next** Punctuation alone doesn't tell you what to write, only how to write it correctly. Chapter 13 goes back to every domain from Part 2 — family, e-commerce, company, library — and shows you, side by side, exactly what Turtle line matches each thing you already built by hand.

Part 3 · Turtle Becomes the Written Form

Chapter 13

From Protégé Tabs to Turtle Lines

Chapters 10 through 12 taught you to read a Turtle file and understand its punctuation. Chapter 13 closes the loop: for every Protégé action you took in Part 2, this chapter shows you the exact Turtle line that same action produces — no new ideas, only translation.

What this chapter produces: four side-by-side pairs — Protégé action next to its Turtle line — covering an individual assertion, a class hierarchy, an inverse-property pair, and a set of data properties.

⚠ **Note:** The snippets below assume the usual rdfs: and owl: prefixes have already been declared, alongside ex: and data:. A complete, loadable version of any of these lines would begin with: @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> . and @prefix owl: <http://www.w3.org/2002/07/owl#> . — the same habit from Chapters 10 and 12, just with one or two more prefixes added.

1. Individuals and object properties — Chapter 5's family tree

Protégé: you created individuals Kabir and Priya, then asserted hasParent → Priya on Kabir.

```
data:kabir ex:hasParent data:priya .
```

One subject, one predicate, one object, one dot. This is the plainest possible Turtle statement — the same shape as every triple since Chapter 1, just written out.

2. Classes and hierarchy — Chapter 6's product catalog

Protégé: Product, with subclass Electronics, with subclass MobilePhone; individuals iPhone15 and GalaxyS24 typed as MobilePhone.

```
ex:Electronics rdfs:subClassOf ex:Product .
ex:MobilePhone rdfs:subClassOf ex:Electronics .

data:iPhone15  a ex:MobilePhone .
data:galaxyS24 a ex:MobilePhone .
```

The lowercase a is shorthand for rdf:type — “this individual belongs to this class.” It is written this way because typing an individual is by far the most common statement in any Turtle file, so Turtle gives it its own short form.

3. Inverse object properties — Chapter 7's company hierarchy

Protégé: manages and reportsTo declared as inverses of each other; Nisha Menon (data:employee_E003) manages Ravi Kumar (data:employee_E002).

```
ex:reportsTo owl:inverseOf ex:manages .

data:employee_E003 ex:manages data:employee_E002 .
```

You only ever assert one direction in the file — manages. The reverse, reportsTo, is implied by the owl:inverseOf declaration, exactly the way Protégé implied it for you without a second click.

4. Data properties — Chapter 8's library catalog

Protégé: Book_TheHobbit with hasTitle, hasPublicationYear, and hasISBN values.

```
data:theHobbit
    ex:hasTitle           "The Hobbit" ;
    ex:hasPublicationYear 1937 ;
    ex:hasISBN            "978-0261102217" .
```

Three data properties, one subject, chained with semicolons — the same punctuation pattern from Chapter 12, applied to a new domain.

📌 **Rule:** Every Turtle line in this chapter is a direct translation of a click you already made in Protégé. If a line ever looks unfamiliar, the question to ask is not “what does this syntax mean” but “which tab did I do this in, back in Part 2?”

Failure story: a beginner, writing Turtle for the first time, invents a brand-new way to represent “is a kind of” instead of reusing rdfs:subClassOf exactly as seen in Protégé's class hierarchy view. The file parses without error, but no reasoner or tool recognises the invented relationship as meaning “is a kind of,” so none of the inference from Chapter 9 — or Part 4 — works on it. The fix: reuse the same handful of standard predicates (a, rdfs:subClassOf, owl:inverseOf) every time these exact situations reappear, rather than improvising new ones.

Try it yourself before checking the answer

Recall Chapter 6's practice: Footwear is a subclass of Product, and RunningShoe is a subclass of Footwear. One individual, data:shoe_001, is typed as RunningShoe. Write the matching Turtle.

Answer key: ex:Footwear rdfs:subClassOf ex:Product . — ex:RunningShoe rdfs:subClassOf ex:Footwear . — data:shoe_001 a ex:RunningShoe . Three short lines, each reusing a predicate already introduced in this chapter.

AI Assist

Ask AI	Here is a Protégé setup in plain English: [describe it in one or two sentences]. Write the matching Turtle lines using only rdf:type (a), rdfs:subClassOf, owl:inverseOf, or the object and data properties already used in this book.
AI may suggest	The correct predicate for each relationship, matching the plain-English description exactly.
You must verify	That the suggested Turtle uses predicates you have actually seen before in this book, rather than an invented one.
Do not accept if	It introduces OWL restrictions, cardinality limits, or reasoning explanations — those belong to Part 4.

Before you leave this chapter

☑ I can write a plain object-property assertion for an individual, without a class.

☑ I can write a subclass chain using rdfs:subClassOf and type an individual using a.

☑ I can write an owl:inverseOf declaration and know only one direction needs to be asserted afterward.

☑ I can write two or three data-property assertions for one individual, chained with semicolons.

→ **Coming up next** Every label you added in Protégé's annotation panel has a Turtle equivalent too. Chapter 14 shows you those lines, and draws a hard line between a comment that helps a human and a rule that a machine actually enforces.

Part 3 · Turtle Becomes the Written Form

Chapter 14

Labels, Definitions, and Comments

In the Part 2 Mini-Lab, you added a label to a class, a label to a property, a label to an individual, and a comment to a class — all inside Protégé's annotation panel. Chapter 14 shows you what those four additions look like as Turtle, and why none of them are rules, no matter how rule-like they sound.

What this chapter produces: Turtle lines for rdfs:label, skos:definition, and rdfs:comment, matched to the four Part 2 Mini-Lab annotations, and one hard rule: an annotation is never an executable constraint.

Annotation	Job	Example
rdfs:label	A short, human-readable display name.	rdfs:label "Leave Request"
skos:definition	A fuller, more formal definition of a term.	skos:definition "A request submitted by an employee to be away from work for a stated period."
rdfs:comment	A free-text note or usage guidance.	rdfs:comment "Use only for active requests; see the capstone for approval history."
Item you already annotated	**What you did in Protégé**	**Turtle line**
Product (class)	Added rdfs:label “Product”	ex:Product rdfs:label "Product" .
hasParent (property)	Added rdfs:label “has parent”	ex:hasParent rdfs:label "has parent" .
Book_TheHobbit (individual)	Added rdfs:label “The Hobbit”	data:theHobbit rdfs:label "The Hobbit" .

Annotation	Job	Example
MobilePhone (class)	Added an rdfs:comment	ex:MobilePhone rdfs:comment "A product category for mobile phones." .

📌 **Rule:** rdfs:label, skos:definition, and rdfs:comment describe a term for a person. None of them are checked by a reasoner, and none of them are enforced on data. Writing rdfs:comment “every leave request must have a status” does not make that true — it only tells a human reader that it should be. Part 6's SHACL is the tool that actually enforces a rule like that.

Pair	Correct separation
rdfs:label vs rdfs:comment	Label is a short display name. Comment is a longer usage note. Neither is enforced.
rdfs:comment vs a SHACL constraint	A comment describes an intention in prose. A SHACL shape (Part 6) actually checks data against it.
skos:definition vs rdfs:comment	Definition states the formal meaning of the term itself. Comment gives usage guidance or a caveat about using the term.

Situation	Your answer
Add a short display name to ex:LeaveRequest for a user interface.	
State, in one formal sentence, what counts as a LeaveRequest, for a glossary.	
Warn future modellers not to reuse ex:hasApprover for a rejected request.	

Answer key: A short display name = rdfs:label. A one-sentence formal statement of meaning = skos:definition. A warning or usage caveat for future modellers = rdfs:comment.

Annotation	Job	Example
🤖 **Ask AI**	Suggest an rdfs:label, a skos:definition, and an rdfs:comment for ex:LeaveRequest, matching the HRMS examples used in this book so far.	
AI may suggest	A short label (“Leave Request”), a one-sentence formal definition, and a short usage note.	
You must verify	That none of the three suggested annotations are phrased as a requirement the data must satisfy — words like “must” or “required” — without a note that this is not enforced.	
Do not accept if	It proposes SHACL syntax, or claims that adding a comment will cause validation to happen automatically.	
☑ I can write rdfs:label, skos:definition, and rdfs:comment for a class, property, or individual. ☑ I can match each of the four Part 2 Mini-Lab annotations to its Turtle line. ☑ I can explain, without hedging, why a comment is not an enforced rule. ☑ I did not write a comment that reads like a validation rule without noting that it isn't one.		

Human-Readable Fact Readings

A fact reading is a reusable sentence pattern showing how a subject relates to another thing or value. It is a fourth documentation job, separate from a short label, formal definition, or usage comment. This book uses a custom annotation property, ex:factReading, to store that pattern.

Declare the custom annotation property once

Because ex:factReading is a custom property created for this book, declare it once as an owl:AnnotationProperty. Standard properties such as rdfs:label, rdfs:comment, and skos:definition do not need to be recreated.

```
ex:factReading a owl:AnnotationProperty ;
    rdfs:label "fact reading"@en ;
    rdfs:comment "Stores a controlled human-readable sentence pattern. It is
documentation only and is not executable."@en .
```

Then add the reading to ex:submittedBy alongside the label, definition, and comment that Chapter 14 already gave it:

```
ex:submittedBy a owl:ObjectProperty ;
    rdfs:label "submitted by"@en ;
    skos:definition "Relates a leave request to the employee who submitted it."@en ;
    rdfs:comment "The target should be an Employee node, not an employee-name
string."@en ;
    ex:factReading "Leave Request is submitted by Employee"@en .
```

A fact reading can describe a datatype property as well as an object property:

```
ex:requestedDays
    ex:factReading "Leave Request has Requested Days"@en .
```

Nothing about the existing labels, definitions, or comments changes. ex:factReading adds one controlled sentence pattern; it does not replace the other annotations.

Beside a specific fact

A property-level fact reading is reusable. You can also place a plain-English companion sentence beside one specific Turtle fact:

```
data:leaveRequest_LR100
    ex:submittedBy data:employee_E001 .
```

Human-readable companion sentence:

```
Leave Request LR100 is submitted by Employee E001.
```

Rule: The companion sentence helps a person read the Turtle fact. In this chapter, it is explanatory prose, not another RDF statement or annotation stored in the graph.

Do not put these in comments or fact readings

Neither rdfs:comment nor ex:factReading executes or enforces anything. Do not use either one to claim:

- Exactly-one, at-most-one, or at-least-one cardinality
- Required-field validation
- Uniqueness
- Query execution
- Workflow actions
- Parser or product compatibility

A reading such as "Each Leave Request must have exactly one submitter" does not enforce that requirement. An executable data check belongs in SHACL.

Mini-exercise

Choose the correct annotation for each situation, now including the fourth documentation job — the reusable fact reading:

Situation	Correct annotation
Give a term a short, human-readable display name	`rdfs:label`
State the formal, precise meaning of a term	`skos:definition`
Record an informal usage note about how a term should be applied	`rdfs:comment`

Show the complete reusable sentence pattern	`ex:factReading`

AI Assist

Do not accept if: It claims ex:factReading is executable, treats the companion sentence as an RDF statement stored in the graph, or says the reading enforces validation, cardinality, querying, or workflow behaviour.

Before you leave this chapter

☐ I can distinguish a short label, formal definition, usage comment, and reusable fact reading.

☐ I know ex:factReading is a custom annotation property created for this book.

☐ I know a property-level fact reading is stored in RDF, while a companion sentence beside one specific triple is explanatory prose.

☐ I know neither a comment nor a fact reading enforces cardinality, validation, querying, or workflow behaviour.

→ **Coming up next** You have now written individuals, properties, classes, and annotations as separate Turtle lines. Chapter 15 returns to a split the Part 2 Mini-Lab already introduced — TBox versus ABox — and shows you, for the first time, what it looks like to keep them in two separate files.

Part 3 · Turtle Becomes the Written Form

Chapter 15

TBox and ABox: Model Terms vs Example Facts

The Part 2 Mini-Lab sorted everything you had built into two piles: reusable vocabulary (TBox) and specific examples (ABox). That was true inside Protégé's tabs. Chapter 15 makes it true on disk too — you will build two small files, load them together in Protégé, and see exactly which file each fact came from.

What this chapter produces: two files — 01_core_tbox.ttl and 02_sample_abox.ttl — loaded together in Protégé, plus a clear habit for deciding which file any new line belongs in.

Same split, new form

A TBox file holds only the reusable vocabulary — the classes and properties that would stay the same even for a completely different set of employees. An ABox file holds only the specific facts — the individuals and their asserted property values. Below is the exact split from the Mini-Lab, written now as two real files.

01_core_tbox.ttl — the reusable vocabulary

```
@prefix ex:   <https://example.org/hrms/ontology/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .

ex:Employee     a owl:Class .
ex:LeaveRequest a owl:Class .
ex:LeaveType    a owl:Class .

ex:submittedBy a owl:ObjectProperty ;
    rdfs:domain ex:LeaveRequest ;
    rdfs:range  ex:Employee .

ex:hasApprover a owl:ObjectProperty ;
    rdfs:domain ex:LeaveRequest ;
    rdfs:range  ex:Employee .

ex:hasLeaveType a owl:ObjectProperty ;
    rdfs:domain ex:LeaveRequest ;
    rdfs:range  ex:LeaveType .

ex:requestedDays a owl:DatatypeProperty ;
    rdfs:domain ex:LeaveRequest ;
    rdfs:range  xsd:integer .

ex:hasStatus a owl:DatatypeProperty ;
    rdfs:domain ex:LeaveRequest ;
    rdfs:range  xsd:string .
```

⚠ **Note:** Do not worry yet about the full meaning of rdfs:domain and rdfs:range. For now, read each pair as a property description: submittedBy is defined for use with LeaveRequest, and its value is expected to be an Employee. This does not yet check that every leave request actually has a submitter — that kind of required-field check is a SHACL job, taught in Part 6. Chapter 17 explains what a reasoner is allowed to infer from domain and range.

02_sample_abox.ttl — the specific example facts

```
@prefix ex:   <https://example.org/hrms/ontology/> .
@prefix data: <https://example.org/hrms/data/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

data:employee_E001 a ex:Employee ;
    rdfs:label "Ananya Rao" .

data:employee_E003 a ex:Employee ;
    rdfs:label "Nisha Menon" .

data:leaveType_CasualLeave a ex:LeaveType ;
    rdfs:label "Casual Leave" .

data:leaveRequest_LR002 a ex:LeaveRequest ;
    ex:submittedBy   data:employee_E001 ;
    ex:hasApprover   data:employee_E003 ;
    ex:hasLeaveType  data:leaveType_CasualLeave ;
    ex:requestedDays 3 ;
    ex:hasStatus     "Pending" .
```

Fact	Which file	Why
submittedBy is defined as a reusable property for LeaveRequest, with its value expected to be an Employee.	01_core_tbox.ttl	True regardless of which employee or request you look at — a property description, not one case.
Ananya Rao's specific leave request is currently Pending.	02_sample_abox.ttl	A fact about one particular request, not a rule about every request.

📌 **Rule:** If a line would still be true after you deleted every employee and leave request in the company, it belongs in the TBox. If a line describes one particular employee or request, it belongs in the ABox.

Loading both files together in Protégé

Appendix A has the exact click path for merging two Turtle files into one active Protégé project. Once both files are loaded together, open the Individuals tab and select data:employee_E001: its type, ex:Employee, came from 01_core_tbox.ttl; its label, “Ananya Rao,” came from 02_sample_abox.ttl. One individual, two files, one merged picture — the same split Part 2's tabs always held, now persisted to disk.

Failure story: a beginner puts a specific fact — “Ananya Rao's leave request is Pending” — directly inside the TBox file, reasoning that “it's important, so it should go in the main file.” Now the vocabulary file is no longer reusable for a different company's data: any new deployment inherits one company's leftover leave request as if it were part of the model itself. The fix: keep the TBox file free of any single business fact, and confirm this by asking, for every line, whether it would still make sense with zero employees in the system.

Try it yourself before checking the answer

Sort each Turtle line into TBox file or ABox file.

Line	TBox or ABox?
ex:LeaveType a owl:Class .	
data:leaveType_CasualLeave a ex:LeaveType .	
ex:requestedDays a owl:DatatypeProperty ; rdfs:range xsd:integer .	
data:leaveRequest_LR003 ex:requestedDays 3 .	

Answer key: ex:LeaveType a owl:Class . — TBox, reusable vocabulary. data:leaveType_CasualLeave a ex:LeaveType . — ABox, one specific leave type instance. ex:requestedDays a owl:DatatypeProperty ; rdfs:range xsd:integer . — TBox, a property definition true for any request. data:leaveRequest_LR003 ex:requestedDays 3 . — ABox, one specific request's value.

How do I know it worked?

Save 01_core_tbox.ttl and 02_sample_abox.ttl, then load both into Protégé (Appendix A has the click path). If both files load without a parser error, your punctuation is valid. Then open the Individuals tab and confirm that data:employee_E001, data:employee_E003, and data:leaveRequest_LR002 all appear, each typed against the class you expect. That confirms the ABox loaded correctly on top of the TBox — the same check works for any TBox/ABox pair in this book, including Checkpoint A.

AI Assist

Ask AI	Here are eight Turtle lines from my HRMS ontology. Sort each into TBox or ABox, and explain your reasoning in one sentence per line.
AI may suggest	A sorted table with short justifications, generally correct if it applies the “would this survive deleting every employee” test.
You must verify	Any line that mixes a class or property declaration with a specific individual assertion — split it into two lines across the two files rather than accepting a mixed line.
Do not accept if	It puts a specific business fact into the TBox file “for convenience,” or merges the two files back into one without you asking for that.

Before you leave this chapter

☑ I can say, in one sentence, what belongs in a TBox file and what belongs in an ABox file.

☑ I built 01_core_tbox.ttl containing only classes and properties.

☑ I built 02_sample_abox.ttl containing only individuals and their asserted facts.

☑ I can load both files together in Protégé and identify which file supplied which part of one individual's picture.

→ **Coming up next** You now have two clean, small files and every skill Part 3 promised — reading, identifying, writing, and splitting Turtle by hand. Checkpoint A asks you to prove it, once, on a fresh HRMS scenario, before Part 4 lets your model start reasoning on its own.

Part 3 · Turtle Becomes the Written Form

Checkpoint A

Assemble and Explain a Tiny TTL Package

This is not a memory test. You will not be asked to write Turtle from a blank page. Checkpoint A gives you a working pattern, asks you to mark its parts, make one small guided change, and explain the result in plain English — the same four skills Part 3 has been building since Chapter 10.

What this checkpoint produces: one TBox file, one ABox file, every line labelled by type, one new individual added by you following the existing pattern, and one short plain-English explanation you write yourself.

The starting package

01_core_tbox.ttl is unchanged from Chapter 15 — the reusable vocabulary doesn't need to grow for this exercise. 02_sample_abox.ttl below adds one more employee, Ravi Kumar, but stops short of giving him a leave request. That is the piece you will add in Task 2.

```
@prefix ex:   <https://example.org/hrms/ontology/> .
@prefix data: <https://example.org/hrms/data/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

data:employee_E001 a ex:Employee ;
    rdfs:label "Ananya Rao" .

data:employee_E002 a ex:Employee ;
    rdfs:label "Ravi Kumar" .

data:employee_E003 a ex:Employee ;
    rdfs:label "Nisha Menon" .

data:leaveType_CasualLeave a ex:LeaveType ;
    rdfs:label "Casual Leave" .

data:leaveRequest_LR002 a ex:LeaveRequest ;
    ex:submittedBy   data:employee_E001 ;
    ex:hasApprover   data:employee_E003 ;
    ex:hasLeaveType  data:leaveType_CasualLeave ;
    ex:requestedDays 3 ;
    ex:hasStatus     "Pending" .
```

Task 1 — Mark every line

For both files, label each declaration or assertion using one of: class declaration, object property declaration, data property declaration, individual + type, object property assertion, data property assertion, or annotation. Use the two tables below as your template.

Line in 01_core_tbox.ttl	Your label
ex:Employee a owl:Class .	
ex:submittedBy a owl:ObjectProperty ; rdfs:domain ... ; rdfs:range	
ex:requestedDays a owl:DatatypeProperty ; rdfs:range xsd:integer .	

Line in 02_sample_abox.ttl	Your label
data:employee_E002 a ex:Employee ;	
rdfs:label "Ravi Kumar" .	
ex:submittedBy data:employee_E001 ;	
ex:requestedDays 3 ;	

Answer key: TBox: ex:Employee a owl:Class . = class declaration. ex:submittedBy ... = object property declaration. ex:requestedDays ... = data property declaration. ABox: data:employee_E002 a ex:Employee ; = individual + type. rdfs:label "Ravi Kumar" . = annotation. ex:submittedBy data:employee_E001 ; = object property assertion. ex:requestedDays 3 ; = data property assertion.

Task 2 — One guided adaptation

Ravi Kumar has also submitted a casual leave request — three days, assigned to Nisha Menon as approver, with status "Approved." Add data:leaveRequest_LR003 to 02_sample_abox.ttl, following the exact pattern already used for data:leaveRequest_LR002. Write your version before checking the answer key.

Answer key:

```
data:leaveRequest_LR003 a ex:LeaveRequest ;
    ex:submittedBy   data:employee_E002 ;
    ex:hasApprover   data:employee_E003 ;
    ex:hasLeaveType  data:leaveType_CasualLeave ;
    ex:requestedDays 3 ;
    ex:hasStatus     "Approved" .
```

Every predicate reuses the same property already declared in the TBox — only the subject and the values changed.

Task 3 — Explain it in plain English

Write three or four plain-English sentences describing what the merged TBox and ABox package now says, as if explaining it to a colleague who has never seen RDF. Check your explanation against the files, sentence by sentence, before comparing it to the model explanation below.

Answer key: This package describes two kinds of things: employees and leave requests, and each request points to a leave type. Ananya Rao's leave request for three days of casual leave is Pending and has Nisha Menon assigned as approver. Ravi Kumar's leave request, for the same three days of casual leave and the same approver, has already been Approved. Nothing here has been checked for correctness yet — Part 6 does that.

📌 **Rule:** Assembling and explaining a small, correct TTL package is a real, checkable skill. Reproducing syntax from blank memory is not what this book asks of you — here, or anywhere else.

AI Assist

🤖 **Ask AI**	Review my TBox and ABox files. Confirm whether Ravi Kumar's new leave request follows the same pattern as Ananya Rao's, and check whether my plain-English explanation matches what the files actually say.
AI may suggest	Pointing out a missing semicolon, a mismatched property, or a plain-English sentence that claims something the files don't actually state — for example, calling a request rejected when the file says Approved.
You must verify	Reread your own explanation against the actual Turtle lines, sentence by sentence, before accepting any AI confirmation that “it matches.”
Do not accept if	It rewrites your explanation for you rather than checking it, or introduces SPARQL or SHACL terminology this checkpoint has not covered yet.

Before you leave this checkpoint

☑ I labelled every line in both files as a class, property, individual, or assertion.

☑ I added one new individual and its assertions, following the existing pattern exactly.

☑ I wrote a plain-English explanation and checked it, sentence by sentence, against the actual files.

☑ I did not try to complete this checkpoint from memory — I used the provided pattern throughout.

→ **Coming up next** Part 3 is complete. You can read, identify, write, and split Turtle by hand, and explain what a small package says. Part 4 asks a new question: now that the model exists, what can it figure out on its own? Chapter 16 opens with the Four Jobs wall chart from Chapter 1, now with real tool names attached, before RDFS and OWL start teaching your graph to reason.

Part 4

Meaning and Reasoning Without Confusion

You are entering Part 4 · Meaning and Reasoning Without Confusion

Your graph can now be written by hand. Part 4 adds only a small amount of new OWL pattern syntax — the main goal is still meaning, not memorising syntax. You will teach your model that one class is a kind of another, that one property is the reverse of another, and that a plain text value can become a controlled node. By the end of Chapter 20, your graph will derive facts you never explicitly wrote, and you will know exactly why a missing fact is not the same as a false one.

Part 4 · Meaning and Reasoning Without Confusion

Chapter 16

Four Jobs You Must Not Mix

Chapter 1 gave four jobs one-word names before you had any tools to attach to them. You now have enough experience with Protégé and Turtle to attach the real tool names — and to see exactly where beginners blend two jobs into one.

What this chapter produces: The Chapter 1 wall chart, reprinted with real tool names attached, and one sorting exercise that catches the most common tool-mixing mistakes before Part 4 begins teaching them one at a time.

The same four jobs, now with tool names

Chapter 1 asked you to keep four jobs separate: model, ask, check, and act. At the time, none of those jobs had a tool attached, because you had not yet met OWL, SPARQL, or SHACL. You have now written Turtle by hand, and you are about to meet the tool that gives your model its first layer of real meaning. Before that happens, the four jobs deserve their tool names.

Job	Plain question it answers	Real tool
Model means	What kinds of things and relationships does the model name, and what do they mean?	OWL and RDFS (Chapters 17–19)
Ask asks	What does the graph currently say — asserted or inferred?	SPARQL (Part 5)
Check checks	Is this data acceptable?	SHACL (Part 6)
Code acts	Now that we know something, what should the computer do about it?	Application logic — outside this book's scope

📌 **Rule:** OWL and RDFS give your model meaning and can infer new facts from it. SPARQL asks about facts that are already in the graph — asserted or inferred. SHACL checks whether data is acceptable. None of the three carries out a business action; that is application logic's job alone, and it stays outside this book.

One sentence to keep: *OWL helps the graph infer meaning. SHACL checks whether the data is acceptable. Everything else in Part 4 is detail on top of that one line.*

Where beginners blend two jobs

Every mistake in Part 4 traces back to blending two of these jobs into one sentence. The three patterns below are worth recognising now, because you will meet a version of each one again before Chapter 20 closes Part 4.

Blended thinking	What actually happens	Correct split
“OWL should reject a leave request with no status.”	OWL reasons under the open-world assumption. A missing fact is unknown, not rejected.	Modelling meaning (OWL) is not the same job as checking acceptability (SHACL).
“I ran a SPARQL query and it proves the data is clean.”	A query only reports what matches. An empty result can mean the problem doesn't exist, or that the query missed it.	Asking (SPARQL) is not the same job as checking (SHACL).
“The ontology should send the approval email.”	Nothing in RDF, OWL, SPARQL, or SHACL sends anything anywhere.	Meaning, asking, and checking are not the same job as acting (application code).

Try it yourself before checking the answer

Sort each sentence into model, ask, check, or act — and name the real tool this time, not just the job.

Sentence	Job + tool
“A LeaveRequest's submittedBy is defined to point to something of type Employee.”	
“Which leave requests currently have no visible approver?”	
“Every submitted leave request must have exactly one hasStatus value.”	
“When a request becomes Approved, notify the employee.”	

Answer: *(1) Model — OWL/RDFS domain and range describe expected types, they don't reject data. (2) Ask — SPARQL, a question about current graph state. (3) Check — SHACL, a required-data rule the data must satisfy. (4) Act — application logic; nothing in this book's four tools sends a notification.*

AI Assist

Ask AI	Using the four jobs and their real tool names, sort these four HRMS sentences. Do not write RDF, SPARQL, or SHACL syntax yet.

AI may suggest	A sorted list mapping each sentence to model/OWL, ask/SPARQL, check/SHACL, or act/application code, with a one-line reason.
You must verify	Check that 'model' answers describe meaning or expected types (not a pass/fail judgement), and that 'check' answers describe a requirement the data must satisfy (not just a type expectation).
Do not accept if	It says OWL 'validates' or 'rejects' anything, or claims a SPARQL result alone proves data quality.

Before you leave this chapter

☑ I can name the real tool behind each of the four jobs: OWL/RDFS, SPARQL, SHACL, application logic.

☑ I can explain why OWL describing a type expectation is not the same as SHACL enforcing a requirement.

☑ I can explain why a SPARQL result is a query answer, not a validation report.

☑ I know none of the four jobs in this book executes a business action.

→ **Coming up next** You now have real tool names attached to all four jobs. Chapter 17 gives you your first reasoning surprise: you'll watch a reasoner add a fact to your graph that you never typed.

Chapter 17 is where the phrase 'the model can figure it out' stops being a promise and starts being something you can actually watch happen.

Part 4 · Meaning and Reasoning Without Confusion

Chapter 17

The Reasoner Adds a Type

You learned to separate model and data across two files. Now watch the model do something Turtle alone never could: add a fact to your graph that you never typed.

What this chapter produces: One moment where you watch a reasoner add a fact you didn't write — seen first in plain English, then in the Turtle that causes it, then in Protégé.

⚠ **Slow lane signpost:** The next four chapters are the steepest climb in this book. Slow down. You are not learning more syntax — you are learning what a graph can mean, what it can infer, and what it still cannot validate. Keep this one split in view the whole way through: OWL and RDFS describe meaning. SHACL checks required data later, in Part 6.

You see this

Ravi Kumar submits a sick leave request. In Protégé, you've marked it as a specific kind of request — a SickLeaveRequest, one level more specific than a plain LeaveRequest. Before any syntax at all, here is the whole idea:

You see: LR005 is a SickLeaveRequest.

Model says: Every SickLeaveRequest is a LeaveRequest.

Reasoner adds: LR005 is also a LeaveRequest — nobody typed this anywhere.

That's it. That's the whole idea of Part 4 in one small example. You'll see this exact moment two more times before Chapter 20 — different facts, same pattern.

The Turtle behind it

Here is the pair of facts that will produce the result you just saw in plain English — one general rule in the TBox, one specific fact in the ABox:

► *Type or copy this exactly*

```
ex:SickLeaveRequest a owl:Class ;
    rdfs:subClassOf ex:LeaveRequest ;
    rdfs:label "Sick Leave Request" .

data:leaveRequest_LR005 a ex:SickLeaveRequest ;
    ex:submittedBy data:employee_E002 ;
    ex:requestedDays 2 .
```

The rule doing the work is rdfs:subClassOf. There's a name for the whole moment — entailment — but the name matters far less than what you already watched happen.

⚠ **Note:** This snippet is lines to add inside your existing 01_core_tbox.ttl and 02_sample_abox.ttl files from Part 3 — not a complete file on its own. It assumes the usual ex:, data:, rdf:, rdfs:, owl:, and xsd: prefixes are already declared, exactly as Chapter 12 taught.

Watch it happen in Protégé

📌 **See it happen: After adding the Turtle lines above and reloading 01_core_tbox.ttl and 02_sample_abox.ttl together in Protégé, start the reasoner (Reasoner menu → the reasoner you already set up in Part 2 → Start reasoner). Select data:leaveRequest_LR005 in the Individuals tab. Look at its Types panel: LeaveRequest now appears there, shown in a different colour or marked as inferred, right alongside the SickLeaveRequest you actually typed. If you only see SickLeaveRequest, check the reasoner is actually running — a stopped reasoner shows only what you typed.**

A second, lighter example: domain

The same kind of surprise happens with rdfs:domain, though it's a smaller idea — worth seeing once, not studying hard. Chapter 15's TBox already says anything used as the subject of ex:submittedBy is a LeaveRequest:

You see: LR004 ex:submittedBy data:employee_E002 . (no type stated on LR004)

Model says: Anything used as the subject of submittedBy is a LeaveRequest.

Reasoner adds: LR004 is a LeaveRequest — inferred, not typed.

Same shape as before: one general rule, one specific fact, one surprise. You don't need to memorise domain and range separately from subclass right now — just notice they produce the same kind of moment.

📌 **Same idea from Part 1, now upgraded:** Chapter 1's tiny graph drew hasStatus pointing at a box labelled ‘Status value: Pending’ — a plain value, not a linked thing. Domain and range can describe what shape that value takes, but they can't turn ‘Pending’ into something the graph reasons about, the way it just reasoned about LR004 and LR005. Chapter 19 makes that upgrade.

Failure story: the subclass that quietly broke a query

A modeller on a different HRMS project needed a fast way to filter Pending requests, so they wrote ex:PendingStatus rdfs:subClassOf ex:LeaveRequest, intending it as a shortcut label for 'requests that are pending.' Every request marked ex:PendingStatus was now also inferred to be a LeaveRequest — harmless on its own. The damage came later: a colleague added a second class, ex:ApprovedStatus, also as a subclass of LeaveRequest, and declared the two disjoint so a request couldn't be both. The next request that moved from Pending to Approved produced a reasoner contradiction, because old data still typed it as both statuses. The subclass was never a kind of request in the first place — it was a status wearing a class's clothing.

Mini-exercise: safe or unsafe subclass?

Apply the all-child-is-parent test: say the sentence 'every X is a Y' out loud. If it's false, the subclass is wrong.

Candidate subclass statement	Safe or unsafe?
ex:SickLeaveRequest rdfs:subClassOf ex:LeaveRequest .	
ex:Manager rdfs:subClassOf ex:Employee .	
ex:PendingStatus rdfs:subClassOf ex:LeaveRequest .	

Answer: *Safe — SickLeaveRequest (every sick leave request is a leave request) and Manager (every manager is an employee, and Nisha Menon already fits this shape). Unsafe — PendingStatus: a status is not a kind of request record, it's a value a request can have.*

AI Assist

Ask AI	Check whether this subclass relationship makes sense, using the all-child-is-parent test.
AI may suggest	It may apply the test correctly to a proposed subclass and flag ones that fail it.
You must verify	For each child class, say the sentence 'every X is a Y' yourself. If it sounds false, reject the subclass regardless of what AI concludes.
Do not accept if	AI treats a status, role, or approval step as a subclass only because the words are loosely related.

Before you leave this chapter

☑ I watched a reasoner add a fact I never typed — in plain English, in Turtle, and in Protégé.

☑ I can apply the all-child-is-parent test before declaring a subclass.

☑ I know domain and range produce the same kind of surprise as subclass, without needing to study them separately yet.

→ **Coming up next** Chapter 17 gave you your first reasoning surprise. Chapter 18 gives you a second, using the inverse link you already wrote in Chapter 13.

You already saw the syntax. Chapter 18 is where you find out what it actually does to your graph.

Part 4 · Meaning and Reasoning Without Confusion

Chapter 18

The Reasoner Flips an Arrow

Chapter 13 had you type a line of Turtle you weren't told the full meaning of yet. Today you find out what it actually does.

What this chapter produces: One reasoning surprise — an arrow firing in the direction you never asserted — seen first in plain English, then confirmed in the Turtle you already wrote back in Chapter 13.

You see this

Chapter 7's company hierarchy lab had Nisha Menon managing Ravi Kumar. You only ever recorded one direction of that relationship. Before any syntax, here's the whole idea:

You wrote: Nisha manages Ravi.

Model says: manages is the reverse of reportsTo.

Reasoner adds: Ravi reportsTo Nisha — nobody typed this arrow anywhere.

Same pattern as Chapter 17: one fact you wrote, one general rule, one fact the reasoner produced on its own. This time the new fact points in the opposite direction from the one you wrote.

The Turtle behind it

Chapter 13 showed you this exact Turtle, as a Protégé-to-Turtle mirror of Chapter 7's lab. At the time, you were only asked to read the syntax:

► *Type or copy this exactly*

```
ex:reportsTo owl:inverseOf ex:manages .

data:employee_E003 ex:manages data:employee_E002 .
```

That single owl:inverseOf line is the whole rule. You asserted manages; the reasoner supplied reportsTo, in the opposite direction, for free.

📌 **Rule:** OWL property meaning derives new facts from facts you already asserted, plus meaning you declared. It cannot invent a business fact that isn't backed by something you wrote — an inverse of nothing is still nothing.

Try it yourself before checking the answer

Given the same inverse pair, read from the other side, and the asserted fact below, what does a reasoner derive?

► *Type or copy this exactly*

```
data:employee_E002 ex:reportsTo data:employee_E003 .
```

You wrote: Ravi reportsTo Nisha.

Reasoner adds: Nisha manages Ravi — the same rule, run the other way.

AI Assist

Ask AI	Review this inverse-property declaration and tell me what a reasoner would derive from one asserted fact.
AI may suggest	The correctly entailed reverse triple, and a plain-English reason.
You must verify	Confirm the inverse pair is declared exactly once, in one direction, and that you are not double-declaring both directions — either alone is sufficient.
Do not accept if	AI introduces an OWL feature you haven't learned yet without asking, or renames manages/reportsTo to something not in your files.

Before you leave this chapter

☑ I watched a reasoner flip an arrow — manages became reportsTo in the opposite direction, using facts I already had.

☑ I predicted the reverse inference correctly before checking the answer.

→ **Coming up next** Chapter 18 gave you a second reasoning surprise: an inverse link. Chapter 19 gives you the biggest payoff yet — turning the plain-text status from Chapter 1 into a real, controlled thing in your graph.

Chapter 19 is where the 'Status value: Pending' box from Chapter 1's very first diagram finally gets upgraded into something the graph can reason about.

Part 4 · Meaning and Reasoning Without Confusion

Chapter 19

Status Becomes a Node

Status was text before. Now you make status a controlled thing — a real node in your graph, not just a word.

What this chapter produces: The status upgrade — from a plain string to a controlled, linked individual — plus a short sorting exercise across the tools you've now met.

The problem with a status that's just text

Chapter 1's very first diagram drew hasStatus pointing at a box labelled ‘Status value: Pending.’ Chapter 15 formalised that as a plain string. Here's the problem with leaving it that way:

Bad for reasoning: hasStatus "Pending" — a piece of text, nothing more.

Better for reasoning: hasStatus data:leaveStatus_Pending — a real node in the graph.

Text is a dead end. “Pending” can't carry a label, can't be linked to anything else, and can't be reasoned about the way data:employee_E001 can. A node can carry all of that. That's the entire upgrade in one sentence — everything below is just how to write it.

Making the upgrade

For data:leaveStatus_Pending to be a real node, it needs to actually exist as an individual, with a class to belong to:

► *Type or copy this exactly*

```
ex:LeaveStatus a owl:Class .

data:leaveStatus_Pending  a ex:LeaveStatus ; rdfs:label "Pending" .
data:leaveStatus_Approved a ex:LeaveStatus ; rdfs:label "Approved" .
data:leaveStatus_Rejected a ex:LeaveStatus ; rdfs:label "Rejected" .

# Replace the old DatatypeProperty declaration from Chapter 15.
# Do not keep both versions of ex:hasStatus — a property cannot
# safely be both a DatatypeProperty and an ObjectProperty at once.
ex:hasStatus a owl:ObjectProperty ;
    rdfs:domain ex:LeaveRequest ;
    rdfs:range ex:LeaveStatus .
```

And every existing assertion changes to match — for LR002, for instance:

► *Before and after, for LR002*

```
# Before (Chapter 15)
data:leaveRequest_LR002 ex:hasStatus "Pending" .

# After (this chapter)
data:leaveRequest_LR002 ex:hasStatus data:leaveStatus_Pending .
```

Optional: closing the list

If you want the model to treat the status class as limited to these three named individuals, one more line closes ex:LeaveStatus to that exact list:

► *Type or copy this exactly*

```
ex:LeaveStatus owl:oneOf ( data:leaveStatus_Pending
                           data:leaveStatus_Approved
                           data:leaveStatus_Rejected ) .
```

This is a nice-to-have, not something to memorise today. The upgrade that matters is the one above: text became a node. Closing the list is a small bonus on top of it.

📌 **Rule:** Turning status into a node — closed list or not — says nothing about whether a given LeaveRequest actually has a status assigned. A request with no hasStatus triple is still simply unknown. Chapter 20 explains why that's the design, not a gap.

Sort these examples: OWL, SHACL, SPARQL, or application logic?

You have now met enough of each tool to sort real examples correctly. Use what each tool actually does — not what the sentence sounds like it's asking for.

Example	Your sort
ex:LeaveStatus is defined as exactly three named individuals.	
"Every submitted LeaveRequest must have a hasStatus value, or the record is rejected."	
"Show me every LeaveRequest with no hasStatus triple."	
"When hasStatus changes to Approved, email the employee."	

Answer: *(1) OWL — the closed list, defining class membership. (2) SHACL — a required-field rule that rejects incomplete data; OWL would only call it unknown. (3) SPARQL — a pattern-matching question over current graph state. (4) Application logic — a triggered action, outside all three graph tools.*

Failure story: assuming the node upgrade also validates

A team building an HRMS prototype made this exact upgrade and assumed it came with a safety net — surely, they thought, a closed list of three statuses means every request must have one. Their demo loaded twelve requests, three of them missing hasStatus entirely, and the reasoner reported zero problems. The team spent an

afternoon suspecting a broken TTL file before realising the upgrade had done exactly what it promised: turn text into a reasoned-about node. It never promised to reject a request with no status at all. The three incomplete requests weren't wrong — they were simply, correctly, unknown. The fix wasn't in the ontology at all; it arrived in Part 6, as a SHACL shape that actually rejects a missing required field.

AI Assist

Ask AI	Sort these four HRMS examples into OWL, SHACL, SPARQL, or application logic, with a one-line reason for each.
AI may suggest	A correctly sorted list distinguishing what the status upgrade means from validation, querying, and action.
You must verify	Check every 'must have, or it's rejected' style sentence — that phrasing almost always means SHACL, not OWL, no matter how it's worded.
Do not accept if	AI says turning status into a node also checks whether a value was actually assigned.

Before you leave this chapter

☑ I upgraded hasStatus from a plain string to a controlled individual, and I can explain why that upgrade matters.

☑ I can sort a real example into OWL, SHACL, SPARQL, or application logic correctly.

☑ I know the status upgrade doesn't reject missing data any more than anything else in Part 4 does.

→ **Coming up next** Chapter 19 gave you the biggest modelling upgrade in Part 4: a plain string became a real, controlled thing. Chapter 20 names the exact reason none of Part 4's tools reject missing data — the open-world assumption — and closes Part 4 with one last recap before Part 5 puts SPARQL in your hands.

Once the open-world assumption fully clicks, SHACL's closed-world checking in Part 6 will feel like the natural next tool, not a contradiction of everything Part 4 just taught you.

Part 4 · Meaning and Reasoning Without Confusion

Chapter 20

Missing Does Not Mean False

Chapter 19's status upgrade didn't reject an incomplete request, and that probably felt wrong. This chapter names the exact reason, so the feeling turns into understanding — and so Part 6's SHACL checking feels like the natural next step rather than a contradiction.

What this chapter produces: One clear rule for why missing data means unknown, one HRMS example you can point to, and a closing recap — sort real facts into asserted, inferred, or unknown — before Part 5 opens with SPARQL.

The one question worth remembering

Statement: LR006 has no submittedBy.

Does OWL say it has no submitter? No.

Does OWL say it is invalid? No.

What does OWL say? Unknown.

Who rejects it later? SHACL, in Part 6.

Everything else in this chapter is that one exchange, explained slowly. If you remember nothing else from Part 4, remember this.

Why missing data is treated as unknown, not wrong

It's natural to expect software to behave like a form screen: leave a required field empty, and the system refuses to submit. OWL does not work that way, and Chapter 19's status upgrade just showed you the consequence. OWL is not checking a submitted form. It is reasoning over a graph that may simply be incomplete.

In a real HRMS, data arrives from several systems at different times. A leave request might exist in the graph before the submitting employee's record has been linked to it — not because anyone made a mistake, but because the two facts arrived in different batches. Under open-world reasoning, that missing link is not automatically false. It is unknown.

This is not a weakness in OWL. It is the entire point of open-world thinking: the graph never pretends it has seen every fact that exists in the world, so it never punishes itself for a fact that simply hasn't arrived yet.

📌 **Rule:** In OWL, missing information means unknown — not false, and not invalid. A validation failure is a judgement about acceptability, and that judgement belongs to SHACL, not to open-world reasoning.

Open-world reasoning vs. the closed-world instinct

Mindset	Plain meaning	HRMS example	Watch out for
Open-world reasoning	The graph may be incomplete. A missing fact may still exist somewhere, just not here yet.	data:leaveRequest_LR006 has no visible ex:submittedBy triple.	Assuming the request definitely has no submitter — that's a closed-world conclusion OWL never draws.
Closed-world checking	If required data is missing at the moment of validation, the record fails.	A shape (Part 6) requires ex:submittedBy on every submitted request — LR006 would fail that check.	Expecting OWL itself to produce this failure. It never will; that job starts in Part 6.

Common mistake: expecting OWL to behave like a form validator

Wrong expectation	Why it fails	Correct thinking
"Missing submittedBy means the request is invalid right now."	OWL's open-world reasoning treats missing as unknown, never as automatically wrong.	The submitter is unknown unless a later validation step requires it and finds it absent.
"OWL should show a user-facing error for this."	OWL is a meaning-and-reasoning layer. It has no concept of a user-facing report.	SHACL produces validation-style reports with focus nodes and messages, starting in Part 6.
"If the graph doesn't say it, it's false."	That's closed-world thinking, and OWL was deliberately not built that way.	The graph may simply be incomplete — not incorrect.

Mini-exercise: unknown, false assumption, or future validation?

Statement	Your bucket	Your reason
LR006 has no visible submittedBy triple.		
Therefore LR006 definitely has no submitter.		
A submitted leave-request file must include submittedBy before acceptance.		
OWL should display: ‘missing requestedDays.’		

Answer: *Open-world unknown — LR006 has no visible submittedBy triple. False assumption — therefore LR006 definitely has no submitter. Future SHACL validation — a submitted file must include submittedBy before acceptance; OWL displaying a missing-field message is the same category error.*

Closing recap: sort these facts

Part 4 taught you three different kinds of true-in-the-graph. Before Part 5 puts SPARQL in your hands to ask questions of this graph, sort each fact below into asserted, inferred, or unknown.

Fact	Asserted / inferred / unknown
data:leaveRequest_LR002 ex:hasStatus data:leaveStatus_Pending .	
data:leaveRequest_LR004 rdf:type ex:LeaveRequest.	
data:employee_E002 ex:reportsTo data:employee_E003.	
Whether data:leaveRequest_LR006 has a submitter at all.	

Answer: *Asserted — you typed the LR002 status triple directly into the file. Inferred — LR004's type came from rdfs:domain on submittedBy (Chapter 17); reportsTo came from owl:inverseOf on manages (Chapter 18). Unknown — LR006's submitter is simply not yet stated anywhere; open-world reasoning does not fill that gap with true or false.*

AI Assist

Ask AI	Classify these missing-data statements as OWL open-world reasoning or future SHACL validation. Do not write SHACL or SPARQL syntax.
AI may suggest	It may separate genuinely unknown facts from statements that describe a future required-field check.
You must verify	Check whether the statement is about what may still turn out to be true, or about a submitted record that must pass a required-field rule.
Do not accept if	AI treats every missing value as false, writes SHACL syntax here, invents a named employee to fill the gap, or claims OWL itself produces a user-facing validation error.

Before you leave this chapter

☑ I can explain that missing data in OWL is unknown, not automatically false.

☑ I can separate open-world reasoning from closed-world data validation.

☑ I can sort a real fact into asserted, inferred, or unknown correctly.

☑ I know that SHACL, starting in Part 6, is the tool that finally closes the world and rejects missing required data.

→ **Part 4 is complete.** You can now give your model real meaning — subclass hierarchies, domain and range, inverse links, and a controlled status — and you know exactly where that meaning stops and validation begins. Part 5 asks a different kind of question: not what does the model mean, but what does the graph currently say? Chapter 21 puts SPARQL in your hands for the first time.

Everything Part 4 taught you about asserted, inferred, and unknown facts is about to become directly queryable. Part 5 is where you start asking your own graph questions and reading its answers back.

Part 5 - Ask the Graph With SPARQL

Opening hook

Part 4 taught you to separate asserted, inferred, and unknown facts by reading triples. That works for a handful of facts. It fails when the graph grows. Part 5 gives you the asking tool: SPARQL.

What Part 5 produces

By the end of this part, you can write small SELECT queries with pattern matching, FILTER, OPTIONAL, COUNT, and ORDER BY. You will also see how a difficult query can expose a modelling mistake.

Rule

SPARQL asks. It does not create facts, fix facts, validate facts, approve workflow, or reject bad data. If you need to reject bad data, wait for SHACL in Part 6.

Part 5 route

Do not treat these three chapters as separate lectures. They are one staircase.

Chapter	Beginner question	What changes
21 - Pattern matching	Which leave requests are pending, and who submitted them?	You learn SELECT and WHERE using two pattern lines.
22 - Practical controls	Can I narrow, allow missing approver, and count?	You add FILTER, OPTIONAL, and COUNT using the same leave-request graph.
23 - Task/Event history	Can the graph answer what happened, when, and by whom?	You use the same asking skill on event history, then hand the next problem to SHACL.

The five SPARQL words you need first

Word	Plain meaning	Beginner warning
PREFIX	Short names for long web addresses.	SPARQL uses PREFIX, not @prefix. No final dot after a PREFIX line.
SELECT	Which blanks you want returned in the answer table.	Only variables listed after SELECT appear in the result.
WHERE	The pattern the graph must match.	Inside WHERE, each pattern line still ends with a dot.
?request	A blank to be filled by the graph.	The same variable name means the same thing across pattern lines.
Result row	One successful match.	Zero rows means no match. It does not automatically mean the data is invalid.

Where to type what

Turtle data goes in a .ttl file and is opened in Protégé. SPARQL query text goes in the SPARQL tab. Do not paste Turtle data into the query box. Do not paste SPARQL SELECT queries into the Turtle file.

Part 5 > Chapter 21

Chapter 21 - SPARQL Pattern Matching

Hook

A human can scan five leave requests and find the pending ones. SPARQL does the same scan for a larger graph, using a pattern instead of eyes.

What this chapter produces

One SELECT query that finds every pending leave request and the employee who submitted it.

Start with the human question

Question

Which leave requests are pending, and who submitted each one?

Before writing syntax, answer the question by looking at a tiny graph. This is not a shortcut. This is how a beginner learns what the query is supposed to match.

Step 1 - Load this tiny leave-request graph

Save this as 02_leave_request_sample.ttl and open it in Protégé. The file is intentionally small. You are not testing memory; you are learning the asking pattern.

```
@prefix ex:    <https://example.org/hrms/ontology/> .
@prefix data:  <https://example.org/hrms/data/> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

# People
 data:employee_E001 a ex:Employee ; rdfs:label "Ananya Rao" .
 data:employee_E002 a ex:Employee ; rdfs:label "Ravi Kumar" .
 data:employee_E003 a ex:Employee ; rdfs:label "Nisha Menon" .

# Status values are nodes, not text strings.
 data:leaveStatus_Approved a ex:LeaveStatus ; rdfs:label "Approved" .
 data:leaveStatus_Pending  a ex:LeaveStatus ; rdfs:label "Pending" .
 data:leaveStatus_Rejected a ex:LeaveStatus ; rdfs:label "Rejected" .

# Leave requests
 data:leaveRequest_LR001 a ex:LeaveRequest ;
     ex:hasStatus data:leaveStatus_Approved ;
     ex:submittedBy data:employee_E001 ;
     ex:requestedDays 3 ;
     ex:hasApprover data:employee_E003 .

 data:leaveRequest_LR003 a ex:LeaveRequest ;
     ex:hasStatus data:leaveStatus_Approved ;
     ex:submittedBy data:employee_E002 ;
     ex:requestedDays 3 ;
     ex:hasApprover data:employee_E003 .

 data:leaveRequest_LR007 a ex:LeaveRequest ;
     ex:hasStatus data:leaveStatus_Pending ;
     ex:submittedBy data:employee_E002 ;
     ex:requestedDays 4 .

 data:leaveRequest_LR008 a ex:LeaveRequest ;
     ex:hasStatus data:leaveStatus_Pending ;
     ex:submittedBy data:employee_E001 ;
     ex:requestedDays 2 ;
     ex:hasApprover data:employee_E003 .

 data:leaveRequest_LR009 a ex:LeaveRequest ;
     ex:hasStatus data:leaveStatus_Rejected ;
     ex:submittedBy data:employee_E001 ;
     ex:requestedDays 5 ;
     ex:hasApprover data:employee_E003 .
```

Step 2 - Answer by eye before writing SPARQL

Look only at ex:hasStatus. Pending appears on LR007 and LR008.

Request	Status	Submitted by	Should appear?
LR001	Approved	employee_E001	No
LR003	Approved	employee_E002	No
LR007	Pending	employee_E002	Yes
LR008	Pending	employee_E001	Yes
LR009	Rejected	employee_E001	No

Step 3 - Turn the question into two pattern lines

Plain English piece	SPARQL pattern line
Find requests whose status is Pending.	?request ex:hasStatus data:leaveStatus_Pending .
For the same request, find who submitted it.	?request ex:submittedBy ?employee .

Rule

The repeated variable ?request is the join. It forces both pattern lines to describe the same leave request. Without that shared variable, you are not asking one connected question.

Step 4 - Run the full query

```
PREFIX ex:   <https://example.org/hrms/ontology/>
PREFIX data: <https://example.org/hrms/data/>

SELECT ?request ?employee
WHERE {
  ?request ex:hasStatus data:leaveStatus_Pending .
  ?request ex:submittedBy ?employee .
}
```

Read it slowly: SELECT says "show me request and employee". WHERE says "find rows where both pattern lines are true".

Expected result

?request	?employee	Human label
data:leaveRequest_LR007	data:employee_E002	Ravi Kumar
data:leaveRequest_LR008	data:employee_E001	Ananya Rao

Beginner note

SPARQL normally returns node IDs such as data:employee_E002. The human label Ravi Kumar is stored in the graph, but this first query does not ask for labels yet. Keep the first query simple.

The beginner mistake that breaks this query

Wrong pattern	Why it fails	Correct pattern
?request ex:hasStatus "Pending" .	"Pending" is text. The graph stores status as a node.	?request ex:hasStatus data:leaveStatus_Pending .

Rule

A text literal cannot match a node. This is not a spelling problem. It is a type-of-value problem.

Try one yourself

Try

Question: Which leave requests did Ravi Kumar submit? Ravi Kumar is data:employee_E002.

You already know the employee, so the blank is only the request.

```
PREFIX ex:   <https://example.org/hrms/ontology/>
PREFIX data: <https://example.org/hrms/data/>

SELECT ?request
WHERE {
  ?request ex:submittedBy data:employee_E002 .
}
```

Expected result

?request
data:leaveRequest_LR003
data:leaveRequest_LR007

AI Assist

Ask AI	AI may suggest	You must verify	Do not accept if
Explain why this pending-leave query returns zero rows even though the graph has pending requests.	It may point to a literal-vs-node mismatch, a misspelled prefix, or a wrong property name.	Check the exact object in the graph: data:leaveStatus_Pending is a node, not the text "Pending".	It says to rewrite the data to match the broken query, or claims SPARQL is broken without checking the pattern.

Before you leave Chapter 21

Checklist

[ ] I can read a WHERE clause as a Turtle-like pattern with blanks.

[ ] I can explain why the same variable connects two lines.

[ ] I know node IDs such as data:leaveStatus_Pending are not the same as text strings such as "Pending".

[ ] I know zero rows means the pattern did not match; it does not automatically prove invalid data.

Coming up next

You can now find rows. Chapter 22 keeps the same graph and adds control: FILTER, OPTIONAL, and COUNT.

Chapter 22 - Practical SPARQL and a Modelling Test

Hook

A plain pattern finds matching rows. Real questions need control: keep only some rows, keep a row even when one link is missing, or count rows instead of listing them.

What this chapter produces

FILTER, OPTIONAL, COUNT, and one modelling test showing why manager-as-text is weak.

Keep the same graph

Do not switch mental worlds. Chapter 22 uses the same leave-request graph from Chapter 21. You are only controlling the answer rows.

One idea: SPARQL first finds rows, then controls them

Control	What it does to rows	Beginner translation
FILTER	Keeps only rows that pass a condition.	Show only requests where days are more than 2.
OPTIONAL	Keeps the row even when one extra link is missing.	Show approver if known; do not hide the request if unknown.
COUNT	Summarises rows instead of listing every row.	Tell me how many requests are in each status.

FILTER - keep rows that pass a condition

Question

Which leave requests ask for more than 2 days?

```
PREFIX ex: <https://example.org/hrms/ontology/>

SELECT ?request ?days
WHERE {
  ?request ex:requestedDays ?days .
  FILTER(?days > 2)
}
```

Expected result

?request	?days
data:leaveRequest_LR001	3
data:leaveRequest_LR003	3
data:leaveRequest_LR007	4
data:leaveRequest_LR009	5

Rule

FILTER can only test ?days because the pattern line above it already found ?days. If you remove the ex:requestedDays line, FILTER has nothing to filter.

OPTIONAL - keep the request even when approver is missing

Question

Show every leave request. If an approver is already assigned, show the approver. If no approver is assigned yet, still show the request.

```
PREFIX ex: <https://example.org/hrms/ontology/>

SELECT ?request ?approver
WHERE {
  ?request a ex:LeaveRequest .
  OPTIONAL { ?request ex:hasApprover ?approver . }
}
```

Expected result

?request	?approver
data:leaveRequest_LR001	data:employee_E003
data:leaveRequest_LR003	data:employee_E003
data:leaveRequest_LR007	(blank / unbound)
data:leaveRequest_LR008	data:employee_E003
data:leaveRequest_LR009	data:employee_E003

The blank approver for LR007 means unknown. It does not mean rejected, wrong, or false. Part 6 later decides whether a missing approver is acceptable.

COUNT and GROUP BY - count rows by status

Question

How many leave requests are in each status?

```
PREFIX ex: <https://example.org/hrms/ontology/>

SELECT ?status (COUNT(?request) AS ?total)
WHERE {
  ?request ex:hasStatus ?status .
}
GROUP BY ?status
```

Expected result

?status	?total
data:leaveStatus_Approved	2
data:leaveStatus_Pending	2
data:leaveStatus_Rejected	1

Rule

GROUP BY means "make one pile per status." COUNT counts the requests inside each pile. If SELECT shows ?status and COUNT(?request), then ?status must be in GROUP BY.

Modelling test - text manager vs linked manager

A fragile query is often exposing a modelling mistake. This is not SPARQL being difficult; it is the graph being weak.

Add this small manager example to your graph.

Cast note

Dev Shah and Leena Iyer are added only for this modelling test. They are temporary example employees used to show how spelling variants split a manager count. The main recurring cast remains Ananya Rao, Ravi Kumar, and Nisha Menon.

```
@prefix ex:    <https://example.org/hrms/ontology/> .
@prefix data:  <https://example.org/hrms/data/> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

# Add this small manager example to the same graph when you reach the modelling test.
 data:employee_E004 a ex:Employee ;
     rdfs:label "Dev Shah" ;
     ex:managerName "N. Menon" ;
     ex:reportsTo data:employee_E003 .

 data:employee_E005 a ex:Employee ;
     rdfs:label "Leena Iyer" ;
     ex:managerName "Nisha Menon" ;
     ex:reportsTo data:employee_E003 .

 data:employee_E002 ex:managerName "Nisha Menon" ;
     ex:reportsTo data:employee_E003 .
```

Bad design: manager stored as text

```
PREFIX ex: <https://example.org/hrms/ontology/>

SELECT ?name (COUNT(?employee) AS ?total)
WHERE {
  ?employee ex:managerName ?name .
}
GROUP BY ?name
```

Expected result

?name	?total
"N. Menon"	1
"Nisha Menon"	2

The count is wrong for the business question. All three employees report to the same real manager, but the text names split into two groups.

Good design: manager stored as a linked node

```
PREFIX ex:   <https://example.org/hrms/ontology/>
PREFIX data: <https://example.org/hrms/data/>

SELECT ?employee
WHERE {
  ?employee ex:reportsTo data:employee_E003 .
}
```

Expected result

?employee
data:employee_E004
data:employee_E005
data:employee_E002

Now the spelling does not matter. All three point to the same node: data:employee_E003. That node can also have labels, department links, roles, or its own manager.

Rule

If the question depends on identity, do not store the answer as text. Store a link to a node. Text is for labels and values. Links are for real relationships.

AI Assist

Ask AI	AI may suggest	You must verify	Do not accept if
Review this query and tell me whether requestedDays, hasStatus, hasApprover, and reportsTo are compared against the right kind of value.	It may flag a literal where a node is expected, a node where a number is expected, or a missing GROUP BY.	Check the graph yourself: requestedDays is a number; hasStatus, hasApprover, and reportsTo point to nodes.	It changes the model to fit a broken query, or treats managerName text as equally reliable identity.

Before you leave Chapter 22

Checklist

[ ] I can use FILTER after a variable has been found in WHERE.

[ ] I can use OPTIONAL when a missing link should not remove the row.

[ ] I can count rows by using COUNT with GROUP BY.

[ ] I can explain why managerName text is weaker than reportsTo node links.

Coming up next

You can now ask controlled questions. Chapter 23 asks a harder business question: not only what is true now, but what happened, when, and who did it.

Chapter 23 - Project Management: Task Node vs Event Node

Hook

A single current-status link can tell you what a task is now. It cannot tell you when it changed or who changed it. For that, the change itself needs a node.

What this chapter produces

A tiny Task/Event model, one overdue-task query, one ordered history query, and a short bridge to SHACL.

First see the modelling problem

Model shape	Can answer current status?	Can answer who changed it and when?
Task has only ex:hasCurrentStatus.	Yes	No. The old status was overwritten.
Task has ex:hasCurrentStatus plus Event nodes.	Yes	Yes. Each change is a separate fact.

Rule

If something must be asked about later, give it its own node. A history fact stored only as overwritten text is already lost.

The tiny model terms

Term	Meaning in this chapter
ex:Task	The work item. Example: Prepare Q3 budget review.
ex:TaskStatus	A node such as Created, In Progress, or Done.
ex:Event	A recorded change that happened to a task.
ex:relatesToTask	Links an Event back to the Task it affected.
ex:performedBy	Links an Event to the Employee who performed it.
ex:occurredAt	Stores when the Event happened.

Step 1 - Load this task/event graph

This is a new tiny lab, but not a new idea. People are still employee nodes. Status is still a node. You are adding event nodes to preserve history.

```
@prefix ex:    <https://example.org/hrms/ontology/> .
@prefix data:  <https://example.org/hrms/data/> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .

 data:employee_E002 a ex:Employee ; rdfs:label "Ravi Kumar" .
 data:employee_E003 a ex:Employee ; rdfs:label "Nisha Menon" .

 data:taskStatus_Created    a ex:TaskStatus ; rdfs:label "Created" .
 data:taskStatus_InProgress a ex:TaskStatus ; rdfs:label "In Progress" .
 data:taskStatus_Done       a ex:TaskStatus ; rdfs:label "Done" .

 data:task_T001 a ex:Task ;
     rdfs:label "Prepare Q3 budget review" ;
     ex:dueDate "2026-07-05"^^xsd:date ;
     ex:hasCurrentStatus data:taskStatus_InProgress .
```

```
data:task_T002 a ex:Task ;
    rdfs:label "Confirm vendor payment file" ;
    ex:dueDate "2026-07-10"^^xsd:date ;
    ex:hasCurrentStatus data:taskStatus_Done .

data:event_EV1 a ex:Event ;
    ex:relatesToTask data:task_T001 ;
    ex:toStatus data:taskStatus_Created ;
    ex:performedBy data:employee_E002 ;
    ex:occurredAt "2026-06-28T09:00:00"^^xsd:dateTime .

data:event_EV2 a ex:Event ;
    ex:relatesToTask data:task_T001 ;
    ex:toStatus data:taskStatus_InProgress ;
    ex:performedBy data:employee_E002 ;
    ex:occurredAt "2026-06-30T14:30:00"^^xsd:dateTime .

data:event_EV3 a ex:Event ;
    ex:relatesToTask data:task_T002 ;
    ex:toStatus data:taskStatus_Created ;
    ex:performedBy data:employee_E003 ;
    ex:occurredAt "2026-07-01T10:00:00"^^xsd:dateTime .

data:event_EV4 a ex:Event ;
    ex:relatesToTask data:task_T002 ;
    ex:toStatus data:taskStatus_Done ;
    ex:performedBy data:employee_E003 ;
    ex:occurredAt "2026-07-02T16:00:00"^^xsd:dateTime .
```

Query 1 - Which tasks are overdue?

Question

On 2026-07-07, which tasks are past their due date and not Done?

```
PREFIX ex:   <https://example.org/hrms/ontology/>
PREFIX data: <https://example.org/hrms/data/>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>

SELECT ?task ?due ?status
WHERE {
  ?task a ex:Task ;
        ex:dueDate ?due ;
        ex:hasCurrentStatus ?status .
  FILTER(?status != data:taskStatus_Done && ?due < "2026-07-07"^^xsd:date)
}
```

Expected result

?task	?due	?status
data:task_T001	2026-07-05	data:taskStatus_InProgress

T001 appears because both halves of the FILTER are true: its due date is before 2026-07-07, and its status is not Done.

T002 does not appear for two independent reasons: its status is Done, and its due date 2026-07-10 has not passed the cutoff.

Query 2 - Show one task history in order

Question

For task_T001, what statuses were recorded, who performed each change, and when did each event happen?

```
PREFIX ex:   <https://example.org/hrms/ontology/>
PREFIX data: <https://example.org/hrms/data/>
```

```
SELECT ?status ?actor ?when
WHERE {
  ?event ex:relatesToTask data:task_T001 ;
         ex:toStatus ?status ;
         ex:performedBy ?actor ;
         ex:occurredAt ?when .
}
ORDER BY ?when
```

Expected result

?status	?actor	?when
data:taskStatus_Created	data:employee_E002	2026-06-28T09:00:00
data:taskStatus_InProgress	data:employee_E002	2026-06-30T14:30:00

Rule

ORDER BY sorts the answer rows. It does not change the data and it does not decide which rows match. It only controls the display order.

Try one yourself

Try

How many events did Ravi Kumar, data:employee_E002, perform?

```
PREFIX ex:   <https://example.org/hrms/ontology/>
PREFIX data: <https://example.org/hrms/data/>

SELECT (COUNT(?event) AS ?total)
WHERE {
  ?event ex:performedBy data:employee_E002 .
}
```

Expected result

?total
2

What SPARQL still cannot do

SPARQL can show you gaps. It can show tasks, events, dates, actors, and missing optional links. But it cannot reject a bad graph. That is the next tool.

Need	Tool
Ask "which tasks are overdue?"	SPARQL
Ask "what happened to task_T001?"	SPARQL
Require every task to have at least one event before accepting the data	SHACL
Send an email when status becomes Done	Application code

Capstone bridge

This Task/Event pattern returns in the capstone as leave-request approval history. The LeaveRequest is the thing being changed. Submission, approval, rejection, and escalation are Events that record what happened, who did it, and when.

Bridge to Part 6

Part 5 helps you notice problems. Part 6 helps you formally check and reject unacceptable data. Do not mix those jobs.

AI Assist

Ask AI	AI may suggest	You must verify	Do not accept if
Check whether my Task/Event model lets me answer both current status and full history.	It may point out missing occurredAt, performedBy, relatesToTask, or toStatus links.	Confirm every Event links to a Task, a status, an actor, and a timestamp. Also check that current status does not silently disagree with latest event.	It replaces Event nodes with one history text field, or claims SPARQL can enforce the required Event rule by itself.

Before you leave Chapter 23

Checklist

[ ] I can explain why a single current-status link cannot answer history questions.

[ ] I can model each change as an Event node.

[ ] I can query overdue tasks using a date and status FILTER.

[ ] I can read ORDER BY as display sorting, not data changing.

[ ] I know SHACL, not SPARQL, is the tool that will require missing data to be fixed.

Coming up next

Part 5 is complete. Every gap you noticed by querying is about to become something you can formally require. Part 6 asks a different question: is this data acceptable?

Part 5 final recap

Do not memorise every query. Memorise the thinking pattern.

When you see this need	Think this
Find matching facts	Write Turtle-shaped lines with variables in WHERE.
Narrow by number, date, or status	Match first, then FILTER.
Keep rows even when a link is missing	Put that extra line inside OPTIONAL.
Summarise rows	Use COUNT and GROUP BY.
A query feels fragile because text spelling matters	The model probably needs a linked node.
You need to reject bad or missing data	Stop. That is SHACL, not SPARQL.

Self-check before moving to SHACL

Can you explain, without looking at code, why LR007 appears in the pending query, why its approver is blank in OPTIONAL, and why SPARQL does not reject that missing approver? If yes, Part 5 has done its job.

Part 6 - Check the Graph With SHACL

Opening hook

Part 5 helped you ask what the graph currently says. That is useful, but it is not enough. A query can show a missing approver; it does not reject the file. Part 6 gives you the checking tool: SHACL.

What Part 6 produces

By the end of this part, you can read and write beginner SHACL shapes for required fields, datatypes, allowed values, relationship targets, and event records. You will also know exactly where SHACL stops: it checks data; it does not run the business process.

Rule

SHACL checks whether a graph is acceptable at validation time. It does not infer missing facts, answer business questions by itself, approve requests, send emails, or replace application code.

Part 6 route

Do not treat these chapters as five unrelated labs. They repeat one loop until it becomes automatic: fail, understand the failure, fix the graph, and rerun validation.

Chapter	Beginner question	What changes
24 - Required fields	Is this leave-request data acceptable?	You learn the first shape: required field, datatype, allowed value.
25 - Traffic values	Can a simple value be limited to allowed choices?	You reuse the same SHACL pattern in a new domain.
26 - Relationship and status checks	Can SHACL check links point to the right kind of node?	You validate object-property targets and recorded status transitions.
27 - Banking event logic	Can event records be checked before a timeline is trusted?	You combine SPARQL timeline thinking with event validation.
28 - Supply chain validation	Can I read a failed report, fix the ABox, and rerun?	You practise the full fail -> fix -> rerun loop.
Checkpoint B	Can I validate a mini graph without help?	TBox + ABox + SHACL + failed report + fix + passing report.

The five SHACL words you need first

Word	Plain meaning	Beginner warning
sh:NodeShape	A check attached to a kind of node.	It is not the business object itself. It is the checker.
sh:targetClass	Which nodes the shape checks.	Only nodes with that rdf:type are checked by this shape.
sh:path	Which property is being checked.	Think: which arrow or value box must exist?
sh:minCount	How many values are required at minimum.	minCount 1 means missing data fails.
sh:in / sh:datatype / sh:class	Allowed values, literal type, or required target class.	Use the right checker for the kind of value you are checking.

Where to type what

TBox and ABox still go in .ttl files. SHACL shapes also go in a .ttl file, usually named with shapes or shacl. You run pySHACL against the data graph using the shapes graph. Do not paste SHACL shapes into the SPARQL query tab and expect a SELECT result.

Beginner safety rail

Do not try to understand a whole SHACL file at once. Read one property block at a time: path -> required count -> value type -> allowed values -> message. If you can explain one property block, you can explain the shape.

File	Beginner job	Do not confuse it with
TBox file	Names the reusable model terms: classes and properties.	It is not the sample data.
ABox file	Contains example facts that may pass or fail.	It is not the rule file.
SHACL shapes file	Contains the accept/reject checks.	It is not a SPARQL query.
Validation report	Explains what failed and where.	It is not the corrected data.

Chapter 24 - Why SHACL Exists + Required Fields

Hook

A missing status was only unknown to OWL and only visible to SPARQL. A submitted file needs a harder judgement: accepted or rejected. That judgement is SHACL.

What this chapter produces

One HRMS shape that checks required status, requestedDays datatype, allowed status values, and submittedBy as an Employee node.

Start with the human check

Question: Is every leave request complete enough to accept into the system? Before syntax, check the tiny graph by eye.

Step 1 - Load this broken leave-request graph

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix data: <https://example.org/hrms/data/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .


data:employee_E001 a ex:Employee ; rdfs:label "Ananya Rao" .
data:employee_E002 a ex:Employee ; rdfs:label "Ravi Kumar" .


data:leaveStatus_Approved a ex:LeaveStatus ; rdfs:label "Approved" .
data:leaveStatus_Pending a ex:LeaveStatus ; rdfs:label "Pending" .
data:leaveStatus_Rejected a ex:LeaveStatus ; rdfs:label "Rejected" .


data:leaveRequest_LR010 a ex:LeaveRequest ;
    ex:hasStatus data:leaveStatus_Pending ;
    ex:submittedBy data:employee_E001 ;
    ex:requestedDays 2 .


data:leaveRequest_LR011 a ex:LeaveRequest ;
    ex:submittedBy data:employee_E002 ;
    ex:requestedDays "three" .


data:leaveRequest_LR012 a ex:LeaveRequest ;
    ex:hasStatus data:leaveStatus_Waiting ;
    ex:submittedBy data:employee_E001 ;
    ex:requestedDays 4 .
```

Step 2 - Decide by eye before writing SHACL

Request	Visible data	Acceptable?	Reason
LR010	status Pending, submittedBy E001, requestedDays 2	Yes	All required values are present and the types look right.
LR011	submittedBy E002, requestedDays "three", no status	No	Status is missing and requestedDays is text, not a whole number.
LR012	status Waiting, submittedBy E001, requestedDays 4	No	Waiting is not one of the allowed status nodes.

Step 3 - Translate the human check into shape pieces

Human rule	SHACL piece
Every LeaveRequest must be checked.	sh:targetClass ex:LeaveRequest
Status must exist once.	sh:path ex:hasStatus with sh:minCount 1 and sh:maxCount 1

Status must be one of three nodes.	sh:in (Pending Approved Rejected)
requestedDays must be a whole number.	sh:datatype xsd:integer
submittedBy must point to an Employee.	sh:class ex:Employee

Step 4 - Run the shape

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix data: <https://example.org/hrms/data/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:LeaveRequestShape a sh:NodeShape ;
    sh:targetClass ex:LeaveRequest ;
    sh:property [
        sh:path ex:hasStatus ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:in ( data:leaveStatus_Pending data:leaveStatus_Approved data:leaveStatus_Rejected ) ;
        sh:message "Leave request must have exactly one allowed status node." ;
    ] ;
    sh:property [
        sh:path ex:requestedDays ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:datatype xsd:integer ;
        sh:minInclusive 1 ;
        sh:message "requestedDays must be a whole number of at least 1." ;
    ] ;
    sh:property [
        sh:path ex:submittedBy ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:class ex:Employee ;
        sh:message "submittedBy must point to one Employee node." ;
    ] .
```

Run it from the folder containing the files:

```
pyshacl -s 03_leave_request_shapes.ttl 02_leave_requests_broken.ttl
```

Command part	Plain meaning
pyshacl	Run the SHACL validator.
-s 03_leave_request_shapes.ttl	Use this file as the shapes/checks graph.
02_leave_requests_broken.ttl	Check this data graph.

Expected validation result, in plain English

Focus node	Problem	Why it fails
data:leaveRequest_LR011	Missing ex:hasStatus	The shape says minCount 1.
data:leaveRequest_LR011	ex:requestedDays is "three"	The shape expects xsd:integer.
data:leaveRequest_LR012	Status data:leaveStatus_Waiting is not allowed	The shape allows only Pending, Approved, or Rejected.

Rule

A validation report is not a query table. It tells you which focus node failed, which path caused the failure, and what constraint was broken.

How to read the validation report without panic

Report word	Beginner meaning	Action

Focus node	The exact data node that failed.	Open that individual/fact block first.
Result path	The property that caused the failure.	Look for that arrow or value in the ABox.
Constraint component	The kind of rule that failed.	Translate it: missing, wrong type, not allowed, too many.
Message	The human explanation written in the shape.	Use it as a hint, then verify the actual triples.

Step 5 - Fix only the broken facts and rerun

```
# Replace the broken LR011 and LR012 facts with this corrected version.
data:leaveRequest_LR011 a ex:LeaveRequest ;
    ex:hasStatus data:leaveStatus_Pending ;
    ex:submittedBy data:employee_E002 ;
    ex:requestedDays 3 .


data:leaveRequest_LR012 a ex:LeaveRequest ;
    ex:hasStatus data:leaveStatus_Pending ;
    ex:submittedBy data:employee_E001 ;
    ex:requestedDays 4 .
```

Expected rerun result: Conforms: True.

The beginner mistake that breaks this chapter

Wrong thought	Why it is wrong	Correct thought
OWL restriction should reject missing status.	OWL reasons under open-world assumptions. Missing is unknown.	SHACL is the closed-world checker.
SPARQL found no missing status, so data is clean.	A query can miss a problem if the pattern is weak.	Validation uses shapes designed to fail bad data.
Status labels are enough.	Labels are text. The model uses status nodes.	Validate the actual node values.

AI Assist

Ask AI	AI may suggest	You must verify	Do not accept if
Explain why LR011 fails this SHACL shape.	It may mention missing hasStatus and requestedDays datatype.	Check the graph yourself: LR011 has no hasStatus and requestedDays is "three".	It says OWL should infer the missing status or that the data is valid because a label exists.

Checklist

[ ] I can explain the difference between asking with SPARQL and checking with SHACL.
[ ] I can read sh:targetClass as the group of nodes being checked.
[ ] I can read sh:minCount 1 as a required field.
[ ] I can explain why a text value fails when xsd:integer is required.
[ ] I know a fixed ABox must be rerun, not just visually inspected.

Coming up next

Chapter 25 repeats the same checking idea in a new setting. This is deliberate. You are not learning a new theory; you are training the validation loop.

Chapter 25 - Traffic Rules: Value Constraints

Hook

Some values are simple choices. A country drives on the left or right. A value like middle should not silently enter the graph.

What this chapter produces

One SPARQL query that asks for left-driving countries and one SHACL shape that rejects missing or invalid drivingSide values.

Same pattern, new setting

Chapter 24 HRMS	Chapter 25 traffic
Leave status must be allowed.	Driving side must be allowed.
Missing status fails.	Missing drivingSide fails.
SHACL checks acceptability.	SHACL checks acceptability.

Step 1 - Load the tiny traffic graph

```
@prefix ex: <https://example.org/traffic/ontology/> .
@prefix data: <https://example.org/traffic/data/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .


data:country_IN a ex:Country ; rdfs:label "India" ; ex:drivingSide "left" .
data:country_US a ex:Country ; rdfs:label "United States" ; ex:drivingSide "right" .
data:country_BAD a ex:Country ; rdfs:label "Bad Example" ; ex:drivingSide "middle" .
data:country_MISSING a ex:Country ; rdfs:label "Missing Side Example" .
```

Step 2 - Ask one SPARQL question first

Question: Which countries drive on the left?

```
PREFIX ex: <https://example.org/traffic/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?country ?label
WHERE {
    ?country a ex:Country ;
        ex:drivingSide "left" ;
        rdfs:label ?label .
}
```

?country	?label
data:country_IN	India

Rule

SPARQL answered the left-driving question. It did not reject country_BAD or country_MISSING. Asking and checking are still different jobs.

Step 3 - Check allowed values with SHACL

```
@prefix ex: <https://example.org/traffic/ontology/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .


ex:CountryShape a sh:NodeShape ;
    sh:targetClass ex:Country ;
    sh:property [
        sh:path ex:drivingSide ;
```

```
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:in ( "left" "right" ) ;
        sh:message "A country drivingSide must be exactly left or right." ;
    ] .
```

Expected validation result

Focus node	Problem	Why it fails
data:country_BAD	drivingSide is "middle"	The shape allows only "left" or "right".
data:country_MISSING	drivingSide is missing	The shape says minCount 1.

When is a literal value acceptable?

A tiny code-like value can stay literal when it does not need its own identity, history, labels in many languages, or relationships. The moment the value needs identity, use a node instead.

Use a literal when...	Use a node when...
The value is a simple code such as "left" or "right".	The value has identity, labels, hierarchy, or relationships.
You only need to compare the exact value.	You need to attach metadata or connect it to other facts.

AI Assist

Ask AI	AI may suggest	You must verify	Do not accept if
Review this traffic example and decide whether drivingSide should be a literal or a node.	It may say literal is enough for two simple values.	Check whether the book needs driving-side identity or only a simple allowed value.	It rewrites every small value as a node without a reason, or claims literals cannot be validated.

Checklist

[ ] I can use sh:in for a small allowed list.
[ ] I can explain why a SPARQL result does not prove the whole file is valid.
[ ] I can decide when a literal value is enough and when a node is better.
[ ] I can connect this chapter back to the required-field pattern from Chapter 24.

Coming up next

Chapter 26 returns to HRMS and checks relationship targets: requester must be an Employee, leave type must be a LeaveType, and status must be a status node.

Chapter 26 - Relationship and Class Constraints + Status That Can Change

Hook

A graph can look connected while still being wrong. A leave request that points submittedBy to the text "Ravi Kumar" is weaker than a request pointing to data:employee_E002. SHACL can catch that.

What this chapter produces

One HRMS relationship-checking shape and one status-change shape. You will also draw the hard boundary: SHACL can check a recorded transition; it does not perform the transition.

Step 1 - Load the broken relationship graph

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix data: <https://example.org/hrms/data/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

data:employee_E001 a ex:Employee ; rdfs:label "Ananya Rao" .
data:employee_E002 a ex:Employee ; rdfs:label "Ravi Kumar" .
data:employee_E003 a ex:Employee ; rdfs:label "Nisha Menon" .

data:leaveType_Annual a ex:LeaveType ; rdfs:label "Annual leave" .
data:leaveType_Sick a ex:LeaveType ; rdfs:label "Sick leave" .

data:leaveStatus_Submitted a ex:LeaveStatus ; rdfs:label "Submitted" .
data:leaveStatus_Pending a ex:LeaveStatus ; rdfs:label "Pending" .
data:leaveStatus_Approved a ex:LeaveStatus ; rdfs:label "Approved" .
data:leaveStatus_Rejected a ex:LeaveStatus ; rdfs:label "Rejected" .
data:leaveStatus_Escalated a ex:LeaveStatus ; rdfs:label "Escalated" .

data:leaveRequest_LR020 a ex:LeaveRequest ;
    ex:submittedBy "Ravi Kumar" ;
    ex:leaveType data:leaveType_Annual ;
    ex:hasStatus data:leaveStatus_Pending .

data:leaveRequest_LR021 a ex:LeaveRequest ;
    ex:submittedBy data:employee_E001 ;
    ex:leaveType data:employee_E003 ;
    ex:hasStatus data:leaveStatus_Pending .

data:statusChange_EV020 a ex:LeaveStatusChange ;
    ex:forRequest data:leaveRequest_LR020 ;
    ex:fromStatus data:leaveStatus_Approved ;
    ex:toStatus data:leaveStatus_Pending ;
    ex:transitionCode "Approved->Pending" .
```

Step 2 - Check the obvious problems by eye

Node	Problem	Correct expectation
LR020	submittedBy is text "Ravi Kumar".	submittedBy should point to an Employee node.
LR021	leaveType points to employee_E003.	leaveType should point to a LeaveType node.
EV020	transitionCode is Approved->Pending.	Only the listed transition codes are accepted.

Step 3 - Run relationship and transition shapes

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
```

```
ex:LeaveRequestRelationshipShape a sh:NodeShape ;
    sh:targetClass ex:LeaveRequest ;
    sh:property [
        sh:path ex:submittedBy ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:class ex:Employee ;
        sh:message "submittedBy must point to an Employee node, not text." ;
    ] ;
    sh:property [
        sh:path ex:leaveType ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:class ex:LeaveType ;
        sh:message "leaveType must point to a LeaveType node." ;
    ] ;
    sh:property [
        sh:path ex:hasStatus ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:class ex:LeaveStatus ;
        sh:message "hasStatus must point to a LeaveStatus node." ;
    ] .

ex:LeaveStatusChangeShape a sh:NodeShape ;
    sh:targetClass ex:LeaveStatusChange ;
    sh:property [
        sh:path ex:forRequest ;
        sh:minCount 1 ;
        sh:class ex:LeaveRequest ;
    ] ;
    sh:property [
        sh:path ex:fromStatus ;
        sh:minCount 1 ;
        sh:class ex:LeaveStatus ;
    ] ;
    sh:property [
        sh:path ex:toStatus ;
        sh:minCount 1 ;
        sh:class ex:LeaveStatus ;
    ] ;
    sh:property [
        sh:path ex:transitionCode ;
        sh:minCount 1 ;
        sh:datatype xsd:string ;
        sh:in ( "Submitted->Pending" "Pending->Approved" "Pending->Rejected" "Pending->Escalated" ) ;
        sh:message "transitionCode must be one of the allowed recorded transitions." ;
    ] .
```

Expected validation result

Focus node	Path	Failure
LR020	ex:submittedBy	Value is a literal, not a node of class ex:Employee.
LR021	ex:leaveType	Value points to an Employee, not a LeaveType.
EV020	ex:transitionCode	Approved->Pending is not in the allowed transition-code list.

What SHACL can and cannot do with status changes

Need	Tool	Reason
Check that a recorded transitionCode is one of the allowed codes.	SHACL	It is a value constraint.
Show the status-change history in time order.	SPARQL	It is a question over Event nodes.
Change the current status from Pending to Approved.	Application code	That is an action, not validation.
Send an approval email.	Application code	SHACL does not trigger emails.

Rule

SHACL can reject a bad recorded fact. It does not create the next fact. Validation is not workflow execution.

Do not overclaim this shape

This beginner shape checks the recorded transitionCode value. It does not calculate the transition from fromStatus and toStatus. More advanced SHACL/SPARQL rules can check that later, but this chapter is deliberately keeping the first status check simple.

Beginner warning: class constraints need type facts

sh:class ex:Employee checks whether the value node is known as an Employee. If your ABox never says data:employee_E002 a ex:Employee, the validator has nothing to confirm. Do not blame SHACL for missing type facts.

AI Assist

Ask AI	AI may suggest	You must verify	Do not accept if
Explain why LR020 and LR021 fail these relationship shapes.	It may identify literal-vs-node and wrong-class target mistakes.	Check the objects of submittedBy and leaveType directly in the graph.	It says labels are enough identity, or says SHACL should approve the workflow.

Checklist

[ ] I can use sh:class when a property must point to a node of a certain class.
[ ] I can explain why a literal employee name fails submittedBy.
[ ] I can check a status-transition value without pretending SHACL runs the workflow.
[ ] I know status history still belongs to Event nodes, not overwritten text.

Coming up next

Chapter 27 uses the same Event pattern in a banking fraud-alert example. The domain changes; the checking loop does not.

Chapter 27 - Banking Fraud Alert: Event-Based Logic Flow

Hook

A fraud alert is not just a label on a transaction. It is an event: it happened at a time, relates to a transaction, has a risk level, and needs a review status.

What this chapter produces

One SPARQL timeline query and one SHACL shape that requires transaction, timestamp, risk level, and review status on each fraud-alert event.

The event pattern you already know

Chapter 23 task event	Chapter 27 fraud event
Event relatesToTask Task	FraudAlertEvent forTransaction Transaction
Event occurredAt time	FraudAlertEvent occurredAt time
Event toStatus status	FraudAlertEvent reviewStatus status
SPARQL orders history	SPARQL orders alert timeline
SHACL checks required event fields	SHACL checks required event fields

Step 1 - Load the banking event graph

```
@prefix ex: <https://example.org/banking/ontology/> .
@prefix data: <https://example.org/banking/data/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

data:transaction_TX100 a ex:Transaction ;
    rdfs:label "Large transfer TX100" ;
    ex:amount 25000 ;
    ex:currency "INR" .

data:transaction_TX101 a ex:Transaction ;
    rdfs:label "Card transaction TX101" ;
    ex:amount 9000 ;
    ex:currency "INR" .

data:fraudEvent_FE1 a ex:FraudAlertEvent ;
    ex:forTransaction data:transaction_TX100 ;
    ex:occurredAt "2026-07-07T09:10:00"^^xsd:dateTime ;
    ex:riskLevel "High" ;
    ex:reviewStatus "Open" .

data:fraudEvent_FE2 a ex:FraudAlertEvent ;
    ex:forTransaction data:transaction_TX101 ;
    ex:riskLevel "Extreme" ;
    ex:reviewStatus "Ignored" .
```

Step 2 - Ask for the visible timeline

```
PREFIX ex: <https://example.org/banking/ontology/>
SELECT ?event ?transaction ?risk ?when
WHERE {
    ?event a ex:FraudAlertEvent ;
        ex:forTransaction ?transaction ;
        ex:riskLevel ?risk ;
        ex:occurredAt ?when .
}
ORDER BY ?when
```

?event	?transaction	?risk	?when
data:fraudEvent_FE1	data:transaction_TX100	High	2026-07-07T09:10:00

Rule

FE2 disappears from this timeline because it has no occurredAt value. That is exactly why a query is not enough. A missing event field can hide the bad row from the query that needs it.

Step 3 - Validate the event records

```
@prefix ex: <https://example.org/banking/ontology/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:FraudAlertEventShape a sh:NodeShape ;
    sh:targetClass ex:FraudAlertEvent ;
    sh:property [
        sh:path ex:forTransaction ;
        sh:minCount 1 ;
        sh:class ex:Transaction ;
    ] ;
    sh:property [
        sh:path ex:occurredAt ;
        sh:minCount 1 ;
        sh:datatype xsd:dateTime ;
    ] ;
    sh:property [
        sh:path ex:riskLevel ;
        sh:minCount 1 ;
        sh:in ( "Low" "Medium" "High" ) ;
    ] ;
    sh:property [
        sh:path ex:reviewStatus ;
        sh:minCount 1 ;
        sh:in ( "Open" "UnderReview" "Cleared" "Escalated" ) ;
    ] .
```

Expected validation result

Focus node	Problem	Why it fails
data:fraudEvent_FE2	Missing occurredAt	Every event needs a timestamp.
data:fraudEvent_FE2	riskLevel is "Extreme"	Allowed values are Low, Medium, High.
data:fraudEvent_FE2	reviewStatus is "Ignored"	Allowed review statuses are Open, UnderReview, Cleared, Escalated.

Fix direction

```
# Keep the event. Fix its values.
data:fraudEvent_FE2 a ex:FraudAlertEvent ;
    ex:forTransaction data:transaction_TX101 ;
    ex:occurredAt "2026-07-07T09:14:00"^^xsd:dateTime ;
    ex:riskLevel "High" ;
    ex:reviewStatus "UnderReview" .
```

AI Assist

Ask AI	AI may suggest	You must verify	Do not accept if
Check whether this fraud-alert event model can answer a timeline query and pass validation.	It may inspect forTransaction, occurredAt, riskLevel, and reviewStatus.	Verify that the query and the shape are doing different jobs.	It says the timeline query proves every event is complete, or removes Event nodes into one text field.

Checklist

[ ] I can reuse the Task/Event pattern outside HRMS.
[ ] I can explain why a missing timestamp can hide a row from a timeline query.
[ ] I can validate required event fields with SHACL.
[ ] I can separate event validation from fraud-decision logic.

Coming up next

Chapter 28 is the strongest validation practice in Part 6: shipment data fails, you read the report, fix the ABox, and rerun.

Chapter 28 - Supply Chain Logistics: Verification and Validation

Hook

A shipment record looks harmless until one missing destination or wrong status feeds a dashboard. SHACL exists to stop bad facts before the graph becomes trusted evidence.

What this chapter produces

One shipment validation shape, one failed report, one fixed ABox block, and one clean rerun.

Step 1 - Load the shipment graph

```
@prefix ex: <https://example.org/supply/ontology/> .
@prefix data: <https://example.org/supply/data/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .


data:location_CHN a ex:Location ; rdfs:label "Chennai warehouse" .
data:location_MAA a ex:Location ; rdfs:label "Main assembly plant" .
data:carrier_C001 a ex:Carrier ; rdfs:label "FastShip Logistics" .


data:shipment_S001 a ex:Shipment ;
    ex:shipmentId "S001" ;
    ex:origin data:location_CHN ;
    ex:destination data:location_MAA ;
    ex:carrier data:carrier_C001 ;
    ex:plannedArrival "2026-07-10"^^xsd:date ;
    ex:shipmentStatus "InTransit" .


data:shipment_S002 a ex:Shipment ;
    ex:shipmentId "S002" ;
    ex:origin data:location_CHN ;
    ex:carrier "FastShip" ;
    ex:plannedArrival "ten July" ;
    ex:shipmentStatus "Flying" .
```

Step 2 - Run the shipment shape

```
@prefix ex: <https://example.org/supply/ontology/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .


ex:ShipmentShape a sh:NodeShape ;
    sh:targetClass ex:Shipment ;
    sh:property [
        sh:path ex:shipmentId ;
        sh:minCount 1 ;
        sh:datatype xsd:string ;
    ] ;
    sh:property [
        sh:path ex:origin ;
        sh:minCount 1 ;
        sh:class ex:Location ;
    ] ;
    sh:property [
        sh:path ex:destination ;
        sh:minCount 1 ;
        sh:class ex:Location ;
    ] ;
    sh:property [
        sh:path ex:carrier ;
        sh:minCount 1 ;
        sh:class ex:Carrier ;
    ] ;
    sh:property [
        sh:path ex:plannedArrival ;
        sh:minCount 1 ;
```

```
        sh:datatype xsd:date ;
    ] ;
    sh:property [
        sh:path ex:shipmentStatus ;
        sh:minCount 1 ;
        sh:in ( "Planned" "InTransit" "Delivered" "Delayed" "Cancelled" ) ;
    ] .
```

Expected failed report

Focus node	Path	Failure
data:shipment_S002	ex:destination	Missing destination.
data:shipment_S002	ex:carrier	Value is text "FastShip", not a Carrier node.
data:shipment_S002	ex:plannedArrival	Value "ten July" is not xsd:date.
data:shipment_S002	ex:shipmentStatus	Value "Flying" is not allowed.

Step 3 - Fix the ABox, not the shape

```
data:shipment_S002 a ex:Shipment ;
    ex:shipmentId "S002" ;
    ex:origin data:location_CHN ;
    ex:destination data:location_MAA ;
    ex:carrier data:carrier_C001 ;
    ex:plannedArrival "2026-07-11"^^xsd:date ;
    ex:shipmentStatus "Delayed" .
```

Rule

Do not weaken the shape to make bad data pass. If the business rule is right, fix the data. Change the shape only when the business rule itself was wrong.

Step 4 - Rerun and record the result

Run	Expected result	What to record
Before fix	Conforms: False	Failed paths and focus nodes.
After fix	Conforms: True	File version, date, and reason for each fix.

Part 6 final recap

When you need this	Think this
Require a value to exist	sh:minCount 1
Allow only one value	sh:maxCount 1
Check a literal datatype	sh:datatype xsd:integer, xsd:date, xsd:dateTime, or xsd:string
Check an allowed list	sh:in (...)
Check a relationship target	sh:class ex:SomeClass
Read a failed report	Focus node + path + failed constraint
Make data pass	Fix the ABox or fix the business rule, then rerun

AI Assist

Ask AI	AI may suggest	You must verify	Do not accept if
Review this shipment validation report and tell me which facts to fix in the ABox.	It may list missing destination, carrier class problem, date datatype, and bad status.	Verify every suggested fix against the shape and the business meaning.	It weakens the shape without explaining why the business rule changed.

Checklist

[ ] I can read a failed validation report without panic.
[ ] I can fix bad ABox facts and rerun validation.
[ ] I can explain why weakening a shape to pass bad data is dangerous.
[ ] I can reuse required-field, datatype, allowed-value, and class constraints across domains.

Checkpoint B - Validate a Mini Graph

Goal

Prove you can run the whole SHACL loop without a new lecture: TBox + ABox + SHACL + failed report + fix + passing report.

Files to create

File	Purpose
01_hrms_tbox.ttl	Names the reusable classes and properties.
02_leave_abox_broken.ttl	Contains sample leave requests, intentionally broken.
03_leave_shapes.ttl	Contains the SHACL checks.
04_validation_notes.md	Records the failed result, the fix, and the passing result.

01_hrms_tbox.ttl

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

ex:Employee a owl:Class ; rdfs:label "Employee" .
ex:Department a owl:Class ; rdfs:label "Department" .
ex:LeaveRequest a owl:Class ; rdfs:label "Leave Request" .
ex:LeaveStatus a owl:Class ; rdfs:label "Leave Status" .

ex:submittedBy a owl:ObjectProperty ;
    rdfs:domain ex:LeaveRequest ;
    rdfs:range ex:Employee .
ex:hasStatus a owl:ObjectProperty ;
    rdfs:domain ex:LeaveRequest ;
    rdfs:range ex:LeaveStatus .
ex:requestedDays a owl:DatatypeProperty ;
    rdfs:domain ex:LeaveRequest .
```

02_leave_abox_broken.ttl

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix data: <https://example.org/hrms/data/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

data:employee_E001 a ex:Employee ; rdfs:label "Ananya Rao" .
data:employee_E002 a ex:Employee ; rdfs:label "Ravi Kumar" .

data:leaveStatus_Pending a ex:LeaveStatus ; rdfs:label "Pending" .
data:leaveStatus_Approved a ex:LeaveStatus ; rdfs:label "Approved" .
data:leaveStatus_Rejected a ex:LeaveStatus ; rdfs:label "Rejected" .

data:leaveRequest_CAP1 a ex:LeaveRequest ;
    ex:hasStatus data:leaveStatus_Pending ;
    ex:requestedDays 2 .

data:leaveRequest_CAP2 a ex:LeaveRequest ;
    ex:submittedBy data:employee_E002 ;
    ex:hasStatus data:leaveStatus_Draft ;
    ex:requestedDays "two" .
```

03_leave_shapes.ttl

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix data: <https://example.org/hrms/data/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
```

```
ex:LeaveRequestShape a sh:NodeShape ;
    sh:targetClass ex:LeaveRequest ;
    sh:property [
        sh:path ex:hasStatus ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:in ( data:leaveStatus_Pending data:leaveStatus_Approved data:leaveStatus_Rejected ) ;
        sh:message "Leave request must have exactly one allowed status node." ;
    ] ;
    sh:property [
        sh:path ex:requestedDays ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:datatype xsd:integer ;
        sh:minInclusive 1 ;
        sh:message "requestedDays must be a whole number of at least 1." ;
    ] ;
    sh:property [
        sh:path ex:submittedBy ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:class ex:Employee ;
        sh:message "submittedBy must point to one Employee node." ;
    ] .
```

Run validation

```
pyshacl -s 03_leave_shapes.ttl 02_leave_abox_broken.ttl
```

Expected failures

Focus node	Failure
data:leaveRequest_CAP1	Missing submittedBy.
data:leaveRequest_CAP2	Status Draft is not in the allowed list.
data:leaveRequest_CAP2	requestedDays "two" is not an integer.

Fix patch

```
data:leaveRequest_CAP1 a ex:LeaveRequest ;
    ex:submittedBy data:employee_E001 ;
    ex:hasStatus data:leaveStatus_Pending ;
    ex:requestedDays 2 .


data:leaveRequest_CAP2 a ex:LeaveRequest ;
    ex:submittedBy data:employee_E002 ;
    ex:hasStatus data:leaveStatus_Approved ;
    ex:requestedDays 2 .
```

Checkpoint pass condition

You pass Checkpoint B only when you can explain the failed report, make the smallest correct ABox fix, rerun validation, and get Conforms: True. Reading the answer is not the same as doing the loop.

Checklist

[ ] I created separate TBox, ABox, and SHACL files.
[ ] I ran pySHACL against the broken ABox.
[ ] I recorded the focus node and path for each failure.
[ ] I fixed the ABox without weakening the shape.
[ ] I reran validation and got Conforms: True.

Part 7 - Build and Maintain the Ontology Project

Opening hook

Parts 1 to 6 taught the pieces. Part 7 teaches the project habit. Most ontology projects fail not because one triple is hard, but because files, tools, versions, and ownership become unclear.

What Part 7 produces

By the end of this part, you can decide which tool should do which job, map a small table or JSON payload into RDF, run a local validation workflow, and maintain a clean change log before the capstone begins.

Rule

A good ontology project is not a giant TTL file. It is a small set of files with clear boundaries, repeatable checks, and written evidence of why changes were made.

Part 7 route

Chapter	Beginner question	What changes
29 - Choosing the right tool	Which tool handles this job?	You lock the model / ask / check / act boundary.
30 - Mapping source data to RDF	How does a row or JSON field become a triple?	You map IDs, labels, links, values, and controlled values deliberately.
31 - Model workflow structure	Can I represent a fixed sequence of steps in TTL?	You add workflow-step individuals, query the next step, and validate the current step - without letting TTL run the workflow.
32 - Governance, versioning, and audit	How do I stop the model becoming unmaintainable?	You add version notes, change logs, AI review discipline, and owner checks.
Checkpoint C	Am I ready for the capstone?	You rehearse TBox + ABox + SHACL + SPARQL with one failure and one fix.

Chapter 29 - Choosing the Right Tool

Hook

Beginners usually do not fail because they forget syntax. They fail because they ask one tool to do another tool's job.

What this chapter produces

A tool-boundary table you can keep beside every ontology project.

The tool-boundary table

Need	Correct tool	Wrong tool to reach for	Plain reason
Name classes and properties.	OWL/RDFS in Turtle or Protégé	SHACL	You are defining the model vocabulary, not checking a submitted file.
See the class tree and properties visually.	Protégé	A giant hand-written TTL file first	Protégé helps beginners see the model before reading lines.
Ask which facts currently match.	SPARQL	SHACL	A question returns rows; it does not produce pass/fail.
Reject missing required fields.	SHACL	OWL restriction	OWL treats missing as unknown; SHACL checks closed-world acceptability.
Send email or update workflow state.	Application code	SHACL or SPARQL	Graph tools do not execute business actions.
Record why a model changed.	Change log + AI Review Log	Memory or chat history	Governance needs explicit evidence in the project folder.

One-minute sorting drill

Sentence	Correct bucket
Every LeaveRequest must have exactly one submittedBy before import.	Check - SHACL
Which requests are currently pending?	Ask - SPARQL
LeaveRequest is a kind of BusinessRecord.	Model - OWL/RDFS
When Approved, notify payroll.	Act - application code
Change status values from text to nodes.	Model/data design decision, then TTL update

Rule

The fastest way to damage this project is to force validation into OWL, workflow into SHACL, or governance into memory. Keep the jobs separate.

Checklist

[ ] I can name the right tool for model, ask, check, and act.
[ ] I can explain why OWL does not reject missing required data.
[ ] I can explain why SPARQL results are not validation reports.
[ ] I know application behavior is outside the ontology files.

Coming up next

Chapter 30 shows how source data becomes RDF. This is where messy tables and JSON payloads become clean graph facts.

Chapter 30 - Mapping Tables, CSV, and JSON to RDF

Hook

A table row is not automatically a good graph. Mapping is where you decide what becomes an IRI, what becomes a label, what becomes a link, and what should stay a literal value.

What this chapter produces

One manual mapping table, one RDF output block, one SPARQL mini-query, and one SHACL mini-check.

Start with a small source table

employeeId	name	managerId	departmentCode
E003	Nisha Menon		HR
E001	Ananya Rao	E003	HR
E002	Ravi Kumar	E003	FIN

Mapping order

Map in this order: stable ID first, class second, label third, relationships fourth, literal values last. Beginners usually start with names. That is the wrong anchor. Names can change; stable IDs should not.

Manual mapping table

Source field	RDF decision	Example
employeeId	IRI identity + employeeId literal	E001 -> data:employee_E001 and ex:employeeId "E001"
name	Human label	rdfs:label "Ananya Rao"
managerId	Object-property link when present	E001 managerId E003 -> data:employee_E001 ex:reportsTo data:employee_E003
departmentCode	Department node link	HR -> data:department_HR

Rule

Stable IDs become IRIs. Human names become labels. Foreign keys become links. Dates and numbers become typed literals. Do not turn everything into text.

RDF output from the mapping

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix data: <https://example.org/hrms/data/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

data:department_HR a ex:Department ; rdfs:label "Human Resources" .
data:department_FIN a ex:Department ; rdfs:label "Finance" .

data:employee_E003 a ex:Employee ;
    ex:employeeId "E003" ;
    rdfs:label "Nisha Menon" ;
    ex:worksIn data:department_HR .

data:employee_E001 a ex:Employee ;
    ex:employeeId "E001" ;
    rdfs:label "Ananya Rao" ;
    ex:reportsTo data:employee_E003 ;
```

```
    ex:worksIn data:department_HR .

data:employee_E002 a ex:Employee ;
    ex:employeeId "E002" ;
    rdfs:label "Ravi Kumar" ;
    ex:reportsTo data:employee_E003 ;
    ex:worksIn data:department_FIN .
```

Mini-query: who reports to Nisha?

```
PREFIX ex: <https://example.org/hrms/ontology/>
PREFIX data: <https://example.org/hrms/data/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?employee ?name
WHERE {
    ?employee ex:reportsTo data:employee_E003 ;
        rdfs:label ?name .
}
ORDER BY ?employee
```

?employee	?name
data:employee_E001	Ananya Rao
data:employee_E002	Ravi Kumar

Mini-check: every imported employee has ID and department

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:EmployeeImportShape a sh:NodeShape ;
    sh:targetClass ex:Employee ;
    sh:property [ sh:path ex:employeeId ; sh:minCount 1 ; sh:datatype xsd:string ] ;
    sh:property [ sh:path ex:worksIn ; sh:minCount 1 ; sh:class ex:Department ] .
```

JSON mapping is the same thinking

JSON shape	RDF mapping decision
{"employeeId":"E001"}	Use the ID to build data:employee_E001.
{"name":"Ananya Rao"}	Store as rdfs:label, not as identity.
{"manager":{"id":"E003"}}	Create a reportsTo link to data:employee_E003.
{"department":"HR"}	Link to data:department_HR, not plain text, if department has identity.

AI Assist

Ask AI	AI may suggest	You must verify	Do not accept if
Convert this employee table to RDF using stable IDs, labels, links, and literals. Explain each mapping decision.	It may create IRIs and map managerId to reportsTo.	Verify every source field: ID, label, link, literal, or ignore.	It uses employee names as IRIs, stores manager as text, or invents missing departments.

Checklist

[ ] I can map a source ID to a stable IRI.
[ ] I can keep a human name as a label, not identity.
[ ] I can turn a foreign key into an object-property link.
[ ] I can write one mini-query and one mini-check for imported data.

Chapter 31 - Model Workflow Structure and Application Boundary

Hook

A leave request does not jump straight from Pending to Approved. It moves through steps: Submission, Manager Review, HR Check, then Approved or Rejected. If those steps only live in someone's head, you cannot ask what happens next, and you cannot catch a request that skipped a step.

What this chapter produces

A workflow-step vocabulary in TTL (Submission, Manager Review, HR Check, Approved, Rejected), one SPARQL query that finds the next step, and one SHACL shape that validates the current step - with a clear boundary: TTL structures the steps; application code runs them.

The five workflow steps as a small TBox

```
@prefix ex:   <https://example.org/hrms/ontology/> .
@prefix data: <https://example.org/hrms/data/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .


ex:WorkflowStep a owl:Class ; rdfs:label "Workflow Step" .
ex:nextStep    a owl:ObjectProperty ; rdfs:domain ex:WorkflowStep ; rdfs:range ex:WorkflowStep .
ex:currentStep a owl:ObjectProperty ; rdfs:domain ex:LeaveRequest ; rdfs:range ex:WorkflowStep .


data:step_Submission    a ex:WorkflowStep ; rdfs:label "Submission" ;
    ex:nextStep data:step_ManagerReview .
data:step_ManagerReview a ex:WorkflowStep ; rdfs:label "Manager Review" ;
    ex:nextStep data:step_HRCheck .
data:step_HRCheck       a ex:WorkflowStep ; rdfs:label "HR Check" ;
    ex:nextStep data:step_Approved, data:step_Rejected .
data:step_Approved      a ex:WorkflowStep ; rdfs:label "Approved" .
data:step_Rejected      a ex:WorkflowStep ; rdfs:label "Rejected" .
```

Why HR Check has two next steps

HR Check points to two next steps on purpose. The graph records that both outcomes are possible; it does not decide which one applies. A person makes that decision, and application code then updates currentStep to record it.

Attach a current step to a leave request

```
data:leaveRequest_LR030 a ex:LeaveRequest ;
    ex:submittedBy data:employee_E001 ;
    ex:currentStep data:step_ManagerReview .
```

SPARQL: what step comes next?

```
PREFIX ex:   <https://example.org/hrms/ontology/>
PREFIX data: <https://example.org/hrms/data/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?request ?nextLabel
WHERE {
    ?request ex:currentStep ?step .
    ?step ex:nextStep ?next .
    ?next rdfs:label ?nextLabel .
}
```

?request	?nextLabel
data:leaveRequest_LR030	HR Check

Read the result correctly

SPARQL only follows the ex:nextStep link that is already stored. It does not decide that the request should move forward, and it does not move it. It answers a question about the current model.

SHACL: is the current step valid?

```
@prefix sh:  <http://www.w3.org/ns/shacl#> .

ex:LeaveRequestStepShape a sh:NodeShape ;
    sh:targetClass ex:LeaveRequest ;
    sh:property [
        sh:path     ex:currentStep ;
        sh:minCount 1 ; sh:maxCount 1 ;
        sh:in ( data:step_Submission data:step_ManagerReview data:step_HRCheck
                data:step_Approved data:step_Rejected ) ;
        sh:message  "currentStep must be exactly one of the five defined workflow steps." ;
    ] .
```

Node	Problem	Reason
data:leaveRequest_LR031	currentStep is missing.	Every leave request must be on a known step.
data:leaveRequest_LR032	currentStep is data:step_InReview.	InReview is not one of the five defined step nodes.

Common mistake: storing the step as free text

Bad move	Why it breaks	Correction
Store the step as ex:currentStep "in review" (a plain string).	Free text cannot be checked against the five known steps, and SPARQL cannot follow it to find the next step.	Use a WorkflowStep individual with rdfs:label, and let SHACL restrict currentStep to the five allowed nodes.

Mini-exercise

Question	Your answer
A request has ex:currentStep data:step_HRCheck. What are the two acceptable next steps, and who decides which one applies?	

Answer

Approved or Rejected. A person (the HR reviewer) decides; the graph only records the outcome after the decision is made, by updating ex:currentStep.

Application boundary for the workflow

Question	Belongs in graph project?	Belongs in app?
What are the five workflow steps?	Yes - WorkflowStep individuals in the TBox.	No.
What step is this request on right now?	Yes - ex:currentStep ABox fact.	App displays it.
Is currentStep one of the five allowed steps?	Yes - SHACL shape.	App may also show a form error, but the shape is the source check.
Move the request from Manager Review to HR Check.	No.	Yes - application code updates ex:currentStep after the manager acts.
Send an email when HR Check finishes.	No.	Yes - application/workflow code.
Escalate if a step sits idle for two days.	No.	Yes - application code with a timer.

Rule

TTL can say what steps exist and which step a request is on. It cannot press Approve, send a notification, or start a timer. Keep the step sequence in the model and the step-changing actions in application code.

AI Assist

Ask AI	AI may suggest	You must verify	Do not accept if
Check this workflow-step model. Does the SPARQL query show the correct next step, and does the shape reject an unknown step name?	It may trace the ex:nextStep chain and check the sh:in list against the five step nodes.	Confirm the shape lists exactly the five step nodes, and that HR Check correctly branches to both Approved and Rejected.	It adds email sending, approval buttons, or timers into the TTL file, or treats SHACL as something that executes the transition rather than checking it.

Practicing this in your project folder

Step	Action	Proof you should have
1 - Write	Add the WorkflowStep individuals, currentStep facts, and the step shape to your files.	A small file change, not a giant rewrite.
2 - Load	Open the TBox + ABox in Protégé.	The five step individuals appear where expected.
3 - Query	Run the next-step SPARQL question.	Expected next-step label matches your by-eye answer.
4 - Validate	Run pySHACL against the step shape.	Failed or passing validation report saved.
5 - Fix	Correct the ABox step value only for a justified reason.	Minimal diff and reason.
6 - Record	Update the change log and AI Review Log.	Future reader can explain what changed and why.

```
# Validate leave requests against the step shape.
pyshacl -s 04_shapes/02_step_shape.ttl 02_abox/02_sample_abox.ttl
```

Checklist

[ ] I can represent a fixed sequence of workflow steps as individuals with rdfs:label.
[ ] I can write a SPARQL query that follows ex:nextStep to find what comes next.
[ ] I can write a SHACL shape that restricts currentStep to the allowed step nodes.
[ ] I can explain why moving a request between steps is application code, not TTL.

Coming up next

Chapter 32 turns this modelled workflow into governance: version notes, change logs, ownership, and audit evidence.

Chapter 32 - Governance, Versioning, and Audit

Hook

An ontology that no one can safely change is not mature. Governance is not paperwork for its own sake; it is how you stop good models from becoming dangerous mystery files.

What this chapter produces

A version note, a change-log pattern, an AI Review Log rule, and an owner checklist.

Version note template

Field	Example
Version	v0.2
Date	2026-07-07
Changed files	01_core_tbox.ttl, 01_required_fields_shapes.ttl
Reason	LeaveRequest must now require exactly one submittedBy before import.
Breaking change?	Yes - old records missing submittedBy now fail validation.
Validation evidence	validation_report_after.txt shows Conforms: True.
Owner approval	HR data owner approved the required-field rule.

Change log pattern

Change type	Example	Minimum evidence
Add class/property	Add ex:LeaveApprovalEvent.	Why needed, label/comment, sample ABox fact.
Change shape	Add sh:minCount 1 on submittedBy.	Business reason, failed-before and passed-after reports.
Rename IRI	Avoid unless necessary.	Migration plan and old-to-new mapping.
Deprecate term	Mark old property as deprecated in comment.	Replacement term and transition plan.
Change allowed values	Add Escalated status.	Business owner approval and tests.

Rule

Stable IDs are not decoration. Renaming an IRI can break queries, shapes, mappings, and application code. Prefer adding a new term with a migration note over casual renaming.

Audit evidence table

Evidence	Why it matters
TBox file version	Shows which vocabulary was used.
ABox sample version	Shows what data was tested.
SHACL shapes version	Shows what acceptability rules were applied.
SPARQL query result	Shows the graph answered the expected question.
Validation report before and after	Shows the exact failure and fix.
AI Review Log	Shows AI suggestions were verified or rejected.
Owner checklist	Shows a human accountable person accepted the change.

Owner checklist

[ ] The business meaning of every new class/property is clear.
[ ] New properties use nodes for identity and literals for values.
[ ] Required fields are enforced in SHACL where needed.
[ ] SPARQL examples still return expected rows.
[ ] Validation passes after the fix.
[ ] Change log explains why the change was made.
[ ] AI Review Log records what was accepted or rejected.

Governance mistakes that look harmless

Mistake	Damage
Changing labels and thinking identity changed.	Labels are human text; IRIs are identity.
Changing IRIs because a label changed.	Breaks links, queries, and shapes.
Letting AI rewrite the model without review.	Can introduce plausible but wrong terms.
Keeping validation reports only in chat.	No future audit evidence.
Mixing test data with production examples.	Beginners cannot tell what is real, sample, or broken.

Checklist

[ ] I can write a version note that explains what changed and why.
[ ] I can protect stable IRIs from casual renaming.
[ ] I can record validation evidence before and after a fix.
[ ] I can use AI as a reviewer, not as the owner of the ontology.

Coming up next

Checkpoint C rehearses the exact three-file pattern you need before the capstone: TBox, ABox, SHACL, plus one query and one fix.

Checkpoint C - Pre-Capstone 3-File Rehearsal

Goal

Before the capstone, prove that you can assemble a tiny package: TBox + ABox + SHACL, then run one SPARQL query, one failed validation, one fix, and one explanation.

File 1 - 01_hrms_tbox.ttl

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

ex:Employee a owl:Class ; rdfs:label "Employee" .
ex:Department a owl:Class ; rdfs:label "Department" .
ex:LeaveRequest a owl:Class ; rdfs:label "Leave Request" .
ex:LeaveStatus a owl:Class ; rdfs:label "Leave Status" .

ex:submittedBy a owl:ObjectProperty ;
    rdfs:domain ex:LeaveRequest ;
    rdfs:range ex:Employee .
ex:hasStatus a owl:ObjectProperty ;
    rdfs:domain ex:LeaveRequest ;
    rdfs:range ex:LeaveStatus .
ex:requestedDays a owl:DatatypeProperty ;
    rdfs:domain ex:LeaveRequest .
```

File 2 - 02_hrms_abox_broken.ttl

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix data: <https://example.org/hrms/data/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

data:employee_E001 a ex:Employee ; rdfs:label "Ananya Rao" .
data:employee_E002 a ex:Employee ; rdfs:label "Ravi Kumar" .

data:leaveStatus_Pending a ex:LeaveStatus ; rdfs:label "Pending" .
data:leaveStatus_Approved a ex:LeaveStatus ; rdfs:label "Approved" .
data:leaveStatus_Rejected a ex:LeaveStatus ; rdfs:label "Rejected" .

data:leaveRequest_CAP1 a ex:LeaveRequest ;
    ex:hasStatus data:leaveStatus_Pending ;
    ex:requestedDays 2 .

data:leaveRequest_CAP2 a ex:LeaveRequest ;
    ex:submittedBy data:employee_E002 ;
    ex:hasStatus data:leaveStatus_Draft ;
    ex:requestedDays "two" .
```

File 3 - 03_hrms_shapes.ttl

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix data: <https://example.org/hrms/data/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:LeaveRequestShape a sh:NodeShape ;
    sh:targetClass ex:LeaveRequest ;
    sh:property [
        sh:path ex:hasStatus ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:in ( data:leaveStatus_Pending data:leaveStatus_Approved data:leaveStatus_Rejected ) ;
        sh:message "Leave request must have exactly one allowed status node." ;
    ] ;
    sh:property [
        sh:path ex:requestedDays ;
        sh:minCount 1 ;
```

```
        sh:maxCount 1 ;
        sh:datatype xsd:integer ;
        sh:minInclusive 1 ;
        sh:message "requestedDays must be a whole number of at least 1." ;
    ] ;
    sh:property [
        sh:path ex:submittedBy ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:class ex:Employee ;
        sh:message "submittedBy must point to one Employee node." ;
    ] .
```

One SPARQL query before validation

Question: Which leave requests are pending, and who submitted them if that link is present?

```
PREFIX ex: <https://example.org/hrms/ontology/>
PREFIX data: <https://example.org/hrms/data/>
SELECT ?request ?employee
WHERE {
    ?request ex:hasStatus data:leaveStatus_Pending .
    OPTIONAL { ?request ex:submittedBy ?employee . }
}
```

?request	?employee
data:leaveRequest_CAP1	(blank / unbound)

Read the result correctly

CAP1 appears because it is Pending. Its employee is blank because submittedBy is missing. SPARQL shows the gap; SHACL rejects it.

Run validation

Why the command uses the ABox here

This mini example repeats type facts inside the ABox, so the validator can find LeaveRequest and Employee nodes directly. In larger projects, you may also pass an ontology/TBox file, but do not add that complexity until this basic loop is clear.

```
pyshacl -s 03_hrms_shapes.ttl 02_hrms_abox_broken.ttl
```

Focus node	Expected failure
data:leaveRequest_CAP1	Missing submittedBy.
data:leaveRequest_CAP2	Status Draft is not allowed.
data:leaveRequest_CAP2	requestedDays "two" is not xsd:integer.

Fix patch

```
data:leaveRequest_CAP1 a ex:LeaveRequest ;
    ex:submittedBy data:employee_E001 ;
    ex:hasStatus data:leaveStatus_Pending ;
    ex:requestedDays 2 .


data:leaveRequest_CAP2 a ex:LeaveRequest ;
    ex:submittedBy data:employee_E002 ;
    ex:hasStatus data:leaveStatus_Approved ;
    ex:requestedDays 2 .
```

Final explanation the reader must be able to give

The TBox names the model terms. The ABox gives sample facts. SPARQL asks which facts currently match. SHACL checks whether the ABox is acceptable. CAP1 showed a missing submittedBy as a blank in SPARQL, then failed SHACL because the shape requires submittedBy. CAP2 failed because Draft was not an allowed status and "two" was not an integer. After fixing the ABox, validation should pass without weakening the shape.

Checkpoint pass condition

You are ready for the capstone only if you can explain this without reading the code aloud line by line. The goal is not memory. The goal is knowing what each file and tool is responsible for.

Checklist

[ ] I can explain the role of each file in the three-file package.
[ ] I can run one SPARQL query and read a blank OPTIONAL value correctly.
[ ] I can run one SHACL validation and read the failed focus nodes.
[ ] I can fix the ABox without weakening the shape.
[ ] I can record the change in a version note and AI Review Log.

Bridge to Part 8

The capstone does not introduce a new world. It combines the company hierarchy, leave-request lifecycle, task/event pattern, SPARQL queries, SHACL checks, and governance notes you have already practised. Part 8 is assembly, not magic.

Part 8 - Capstone: The HRMS Leave Request Engine

Opening hook
The capstone is not a new theory chapter. It is the moment where the pieces stop living separately. Company hierarchy, leave requests, task/event history, SPARQL questions, SHACL checks, and governance records now become one small ontology project.

What Part 8 produces
By the end, you can build a tiny HRMS leave-request package with TBox, ABox, SPARQL, SHACL, validation notes, an AI Review Log, and an owner checklist. You will also be able to explain why each file exists.

Rule
Do not treat the capstone as memory work. The goal is to know which file owns which responsibility and to prove the graph with query and validation evidence.

Part 8 route

Chapter	Beginner question	What changes
33 - HRMS Leave Request Engine	Can I combine people, roles, tasks, events, queries, checks, and governance?	You assemble one small project instead of learning another isolated concept.
Capstone proof	Can I show one failure and one fix without weakening the model?	You run SPARQL to observe, SHACL to reject, ABox fixes to repair, and governance notes to record.

Checklist
[ ] I know this capstone reuses earlier ideas instead of introducing a new domain.
[ ] I can point to the model, ask, check, act, and govern boundaries.
[ ] I know the app is not being built here; the graph package is being proven first.

Chapter 33 - People, Roles, Tasks, and Events Together

Hook
A leave request is not just one row. It has a requester, a current status, a review task, a responsible role, and a history of events. If you model only the row, you lose the workflow evidence.

What this chapter produces
One capstone package: TBox, broken ABox, three SPARQL queries, SHACL shapes, failed validation notes, a fix patch, and governance evidence.

The capstone story in plain English

Swatantra.ai has HR and Finance departments. Employees belong to departments and hold roles. An employee submits a leave request. The request creates a review task assigned to a role, not directly to a person. Events record what happened over time.

Piece	Plain meaning	Why it exists in the capstone
Department	Where the employee belongs.	Reuses the company hierarchy idea from the visual chapters.
Employee	The person who submits or performs work.	Gives the graph real actors.
Role	The responsibility bucket.	Tasks should be assigned to a role before an app chooses a person.
LeaveRequest	The business record.	The central object being queried and validated.
LeaveTask	The work item created for review.	Shows work that must be assigned and tracked.
LeaveEvent	The timestamped history record.	Preserves status changes instead of overwriting history.
SHACL shape	The accept/reject checker.	Rejects bad data without pretending to run the workflow.

Visual map before files

```
Organization
  -> hasDepartment -> Department

Employee
  -> memberOfDepartment -> Department
  -> holdsRole -> Role

LeaveRequest
  -> submittedBy -> Employee
  -> hasStatus -> LeaveStatus
  -> requestedDays -> integer value

LeaveTask
  -> forRequest -> LeaveRequest
  -> assignedToRole -> Role
  -> taskStatus -> TaskStatus
  -> dueDate -> date value

LeaveEvent
  -> forRequest -> LeaveRequest
  -> performedBy -> Employee
  -> toStatus -> LeaveStatus
  -> occurredAt -> dateTime value
```

Beginner warning
The task points to a Role, not directly to Nisha. That is deliberate. The model says what responsibility is needed. The application or HR process can later decide which person currently fills that role.

Checklist
[ ] I can explain why a leave request needs both current status and event history.
[ ] I can explain why a review task points to a role instead of a person.

[ ] I can read the diagram as triples before opening the TTL files.

Capstone project folder

Create the package as separate files. A beginner project becomes confusing when every idea is thrown into one giant file.

```
hrms-leave-capstone/
  01_tbox/
    01_hrms_capstone_tbox.ttl
  02_abox/
    02_hrms_capstone_abox_broken.ttl
    02_hrms_capstone_abox_fixed.ttl
  03_queries/
    01_pending_leave_requests.sparql
    02_tasks_for_hr_reviewer.sparql
    03_event_history_for_request.sparql
  04_shapes/
    04_hrms_capstone_shapes.ttl
  05_reports/
    validation_report_before.txt
    validation_report_after.txt
    sparql_expected_results.md
  06_governance/
    CHANGELOG.md
    AI_REVIEW_LOG.md
    OWNER_CHECKLIST.md
  README.md
```

File	Beginner job	Do not confuse it with
TBox	Names reusable classes and properties.	It is not sample HRMS data.
ABox	Contains sample facts that may pass or fail.	It is not the rule file.
SPARQL	Asks questions over the visible graph.	It is not validation.
SHACL	Checks whether the graph is acceptable.	It is not workflow code.
Governance notes	Record what changed and why.	They are not optional memory.

Rule

If a future reader cannot tell which file to open for model terms, sample facts, questions, checks, and decisions, the project is already drifting.

File 1 - 01_hrms_capstone_tbox.ttl

This file names the vocabulary. It does not decide whether today's sample leave request is valid.

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

ex:Organization a owl:Class ;
    rdfs:label "Organization" ;
    rdfs:comment "The company or legal unit in the HRMS graph." .

ex:Department a owl:Class ;
    rdfs:label "Department" ;
    rdfs:comment "A business department such as HR or Finance." .

ex:Role a owl:Class ;
    rdfs:label "Role" ;
    rdfs:comment "A responsibility bucket used for task assignment." .

ex:Employee a owl:Class ;
    rdfs:label "Employee" ;
    rdfs:comment "A person recorded as an employee in the sample HRMS graph." .

ex:LeaveRequest a owl:Class ;
    rdfs:label "Leave Request" ;
    rdfs:comment "A request for leave submitted by an employee." .

ex:LeaveStatus a owl:Class ;
```

```
    rdfs:label "Leave Status" ;
    rdfs:comment "A controlled status node for a leave request." .

ex:LeaveTask a owl:Class ;
    rdfs:label "Leave Task" ;
    rdfs:comment "A work item created to review or process a leave request." .

ex:TaskStatus a owl:Class ;
    rdfs:label "Task Status" ;
    rdfs:comment "A controlled status node for a task." .

ex:LeaveEvent a owl:Class ;
    rdfs:label "Leave Event" ;
    rdfs:comment "A timestamped event in the leave-request history." .
```

```
ex:hasDepartment a owl:ObjectProperty ;
    rdfs:label "has department" ;
    rdfs:comment "Connects an organization to a department." .

ex:memberOfDepartment a owl:ObjectProperty ;
    rdfs:label "member of department" ;
    rdfs:comment "Connects an employee to a department." .

ex:holdsRole a owl:ObjectProperty ;
    rdfs:label "holds role" ;
    rdfs:comment "Connects an employee to a role." .

ex:submittedBy a owl:ObjectProperty ;
    rdfs:label "submitted by" ;
    rdfs:comment "Connects a leave request to the employee who submitted it." .

ex:hasStatus a owl:ObjectProperty ;
    rdfs:label "has status" ;
    rdfs:comment "Connects a leave request to its current status node." .

ex:forRequest a owl:ObjectProperty ;
    rdfs:label "for request" ;
    rdfs:comment "Connects a task or event to the leave request it belongs to." .

ex:assignedToRole a owl:ObjectProperty ;
    rdfs:label "assigned to role" ;
    rdfs:comment "Connects a task to the role responsible for the task." .

ex:taskStatus a owl:ObjectProperty ;
    rdfs:label "task status" ;
    rdfs:comment "Connects a task to its current task status node." .

ex:performedBy a owl:ObjectProperty ;
    rdfs:label "performed by" ;
    rdfs:comment "Connects an event to the employee who performed it." .

ex:toStatus a owl:ObjectProperty ;
    rdfs:label "to status" ;
    rdfs:comment "Connects a leave event to the resulting status." .
ex:fromStatus a owl:ObjectProperty ;
    rdfs:label "from status" ;
    rdfs:comment "Connects a leave event to the status it changed from." .
ex:transitionCode a owl:DatatypeProperty ; rdfs:label "transition code" ; rdfs:comment "A recorded from-to status code for the
event, checked against the allowed transition list." .

ex:employeeId a owl:DatatypeProperty ; rdfs:label "employee ID" .
ex:requestId a owl:DatatypeProperty ; rdfs:label "request ID" .
ex:taskId a owl:DatatypeProperty ; rdfs:label "task ID" .
ex:eventId a owl:DatatypeProperty ; rdfs:label "event ID" .
ex:requestedDays a owl:DatatypeProperty ; rdfs:label "requested days" .
ex:taskName a owl:DatatypeProperty ; rdfs:label "task name" .
ex:dueDate a owl:DatatypeProperty ; rdfs:label "due date" .
ex:eventType a owl:DatatypeProperty ; rdfs:label "event type" .
ex:occurredAt a owl:DatatypeProperty ; rdfs:label "occurred at" .
```

TBox term	Beginner reading
ex:LeaveRequest	A reusable class name for leave-request records.
ex:submittedBy	A relationship from a leave request to an employee.
ex:assignedToRole	A relationship from a task to a role.

ex:requestedDays	A value field, later checked as an integer.
rdfs:comment	Human explanation, not an executable rule.

File 2 - 02_hrms_capstone_abox_broken.ttl

This ABox is intentionally broken. The point is not to hide mistakes. The point is to prove that the project can find and fix them.

People, departments, roles, and status nodes

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix data: <https://example.org/hrms/data/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .


data:org_Swatantra a ex:Organization ;
    rdfs:label "Swatantra.ai" ;
    ex:hasDepartment data:dept_HR, data:dept_FIN .


data:dept_HR a ex:Department ; rdfs:label "Human Resources" .
data:dept_FIN a ex:Department ; rdfs:label "Finance" .


data:role_Employee a ex:Role ; rdfs:label "Employee" .
data:role_Manager a ex:Role ; rdfs:label "Manager" .
data:role_HRReviewer a ex:Role ; rdfs:label "HR Reviewer" .


data:employee_E001 a ex:Employee ;
    ex:employeeId "E001" ;
    rdfs:label "Ananya Rao" ;
    ex:memberOfDepartment data:dept_HR ;
    ex:holdsRole data:role_Employee .


data:employee_E002 a ex:Employee ;
    ex:employeeId "E002" ;
    rdfs:label "Ravi Kumar" ;
    ex:memberOfDepartment data:dept_FIN ;
    ex:holdsRole data:role_Employee .


data:employee_E003 a ex:Employee ;
    ex:employeeId "E003" ;
    rdfs:label "Nisha Menon" ;
    ex:memberOfDepartment data:dept_HR ;
    ex:holdsRole data:role_Manager, data:role_HRReviewer .


data:leaveStatus_Pending a ex:LeaveStatus ; rdfs:label "Pending" .
data:leaveStatus_Approved a ex:LeaveStatus ; rdfs:label "Approved" .
data:leaveStatus_Rejected a ex:LeaveStatus ; rdfs:label "Rejected" .
data:leaveStatus_Escalated a ex:LeaveStatus ; rdfs:label "Escalated" .


data:taskStatus_Open a ex:TaskStatus ; rdfs:label "Open" .
data:taskStatus_Done a ex:TaskStatus ; rdfs:label "Done" .
```

Leave requests, with two intentional problems

```
data:leaveRequest_LR100 a ex:LeaveRequest ;
    ex:requestId "LR100" ;
    ex:submittedBy data:employee_E001 ;
    ex:hasStatus data:leaveStatus_Pending ;
    ex:requestedDays 3 .


data:leaveRequest_LR101 a ex:LeaveRequest ;
    ex:requestId "LR101" ;
    ex:hasStatus data:leaveStatus_Pending ;
    ex:requestedDays "two" .


data:leaveRequest_LR102 a ex:LeaveRequest ;
    ex:requestId "LR102" ;
    ex:submittedBy data:employee_E002 ;
    ex:hasStatus data:leaveStatus_Draft ;
    ex:requestedDays 2 .
```

Review tasks, with two intentional problems

```
data:task_T100 a ex:LeaveTask ;
    ex:taskId "T100" ;
    ex:taskName "Review LR100" ;
```

```
    ex:forRequest data:leaveRequest_LR100 ;
    ex:assignedToRole data:role_HRReviewer ;
    ex:taskStatus data:taskStatus_Open ;
    ex:dueDate "2026-07-09"^^xsd:date .

data:task_T101 a ex:LeaveTask ;
    ex:taskId "T101" ;
    ex:taskName "Review LR101" ;
    ex:forRequest data:leaveRequest_LR101 ;
    ex:taskStatus data:taskStatus_Open ;
    ex:dueDate "2026-07-09"^^xsd:date .

data:task_T102 a ex:LeaveTask ;
    ex:taskId "T102" ;
    ex:taskName "Review LR102" ;
    ex:forRequest data:leaveRequest_LR102 ;
    ex:assignedToRole data:employee_E003 ;
    ex:taskStatus data:taskStatus_Open ;
    ex:dueDate "2026-07-09"^^xsd:date .
```

Event history, with two intentional problems

```
data:event_EV100 a ex:LeaveEvent ;
    ex:eventId "EV100" ;
    ex:eventType "Submitted" ;
    ex:forRequest data:leaveRequest_LR100 ;
    ex:performedBy data:employee_E001 ;
    ex:toStatus data:leaveStatus_Pending ;
    ex:occurredAt "2026-07-07T09:00:00"^^xsd:dateTime .

data:event_EV101 a ex:LeaveEvent ;
    ex:eventId "EV101" ;
    ex:eventType "Reviewed" ;
    ex:forRequest data:leaveRequest_LR102 ;
    ex:toStatus data:leaveStatus_Draft ;
    ex:occurredAt "2026-07-07T10:30:00"^^xsd:dateTime .

data:event_EV103 a ex:LeaveEvent ;
    ex:eventId "EV103" ;
    ex:eventType "Reviewed" ;
    ex:forRequest data:leaveRequest_LR100 ;
    ex:performedBy data:employee_E003 ;
    ex:fromStatus data:leaveStatus_Approved ;
    ex:toStatus data:leaveStatus_Pending ;
    ex:transitionCode "Approved->Pending" ;
    ex:occurredAt "2026-07-07T11:15:00"^^xsd:dateTime .
```

By-eye inspection before tools

Node	Visible problem	What a correct graph should have
LR101	No submittedBy and requestedDays is text "two".	One Employee link and an integer day count.
LR102	Status points to Draft, which is not an allowed status node.	One allowed status: Pending, Approved, Rejected, or Escalated.
T101	Review task has no assignedToRole.	Every task must have one responsible Role.
T102	assignedToRole points to employee_E003.	assignedToRole must point to a Role node, not a person.
EV101	Event has no performedBy and points to Draft status.	Every event needs an actor and an allowed resulting status.
EV103	transitionCode is Approved->Pending, which is not an allowed transition.	Only listed transitions are accepted (for example Pending->Approved or Pending->Rejected).

Rule
Always inspect the small graph by eye before running tools. Tools are proof, not a replacement for understanding.

SPARQL: ask what the broken graph currently says

Boundary
The three queries below are not validation. They show what is visible. Some broken records may still appear, and some broken records may disappear from a query because required fields are missing.

Query 1 - Pending leave requests

```
PREFIX ex: <https://example.org/hrms/ontology/>
PREFIX data: <https://example.org/hrms/data/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?request ?employeeName ?days
WHERE {
    ?request a ex:LeaveRequest ;
             ex:hasStatus data:leaveStatus_Pending ;
             ex:requestedDays ?days .
    OPTIONAL {
        ?request ex:submittedBy ?employee .
        ?employee rdfs:label ?employeeName .
    }
}
ORDER BY ?request
```

?request	?employeeName	?days
data:leaveRequest_LR100	Ananya Rao	3
data:leaveRequest_LR101	(blank / unbound)	two

Read this result correctly
LR101 appears because it is Pending. The employee is blank because submittedBy is missing. The days value appears as text. SPARQL shows the problem; it does not reject the file.

Query 2 - Tasks assigned to the HR Reviewer role

```
PREFIX ex: <https://example.org/hrms/ontology/>
PREFIX data: <https://example.org/hrms/data/>

SELECT ?task ?request ?name
WHERE {
    ?task a ex:LeaveTask ;
          ex:assignedToRole data:role_HRReviewer ;
          ex:forRequest ?request ;
          ex:taskName ?name .
}
ORDER BY ?task
```

?task	?request	?name
data:task_T100	data:leaveRequest_LR100	Review LR100

Read this result correctly
T101 is missing assignedToRole, so it does not appear. T102 points to an employee instead of the HR Reviewer role, so it also does not appear. The query is useful, but it is still not a validation report.

Query 3 - Event history for LR100

```
PREFIX ex: <https://example.org/hrms/ontology/>
PREFIX data: <https://example.org/hrms/data/>

SELECT ?event ?eventType ?actor ?toStatus ?when
WHERE {
    ?event a ex:LeaveEvent ;
           ex:forRequest data:leaveRequest_LR100 ;
           ex:eventType ?eventType ;
```

```
          ex:performedBy ?actor ;
          ex:toStatus ?toStatus ;
          ex:occurredAt ?when .
}
ORDER BY ?when
```

?event	?eventType	?actor	?toStatus	?when
data:event_EV100	Submitted	data:employee_E001	data:leaveStatus_Pending	2026-07-07T09:00:00

Checklist

[ ] I can read blank OPTIONAL values without panicking.
[ ] I can explain why a query result does not prove the graph is valid.
[ ] I can use queries as evidence of what the graph currently says.

File 4 - 04_hrms_capstone_shapes.ttl

The SHACL file checks acceptability. It does not approve a leave request, send email, or assign a real person to the task.

Shape 1 - Leave requests

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix data: <https://example.org/hrms/data/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:LeaveRequestShape a sh:NodeShape ;
    sh:targetClass ex:LeaveRequest ;
    sh:property [
        sh:path ex:submittedBy ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:class ex:Employee ;
        sh:message "Each leave request must point to one Employee." ;
    ] ;
    sh:property [
        sh:path ex:hasStatus ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:in ( data:leaveStatus_Pending data:leaveStatus_Approved
                data:leaveStatus_Rejected data:leaveStatus_Escalated ) ;
        sh:message "Leave status must be one allowed status node." ;
    ] ;
    sh:property [
        sh:path ex:requestedDays ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:datatype xsd:integer ;
        sh:minInclusive 1 ;
        sh:message "requestedDays must be a whole number of at least 1." ;
    ] .
```

Shape 2 - Leave tasks

```
ex:LeaveTaskShape a sh:NodeShape ;
    sh:targetClass ex:LeaveTask ;
    sh:property [
        sh:path ex:forRequest ;
        sh:minCount 1 ;
        sh:class ex:LeaveRequest ;
    ] ;
    sh:property [
        sh:path ex:assignedToRole ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:class ex:Role ;
        sh:message "Tasks must be assigned to a Role node, not a person." ;
    ] ;
    sh:property [
        sh:path ex:taskStatus ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:in ( data:taskStatus_Open data:taskStatus_Done ) ;
    ] ;
    sh:property [
        sh:path ex:dueDate ;
        sh:minCount 1 ;
        sh:datatype xsd:date ;
    ] .
```

Shape 3 - Leave events

```
ex:LeaveEventShape a sh:NodeShape ;
    sh:targetClass ex:LeaveEvent ;
    sh:property [
        sh:path ex:forRequest ;
        sh:minCount 1 ;
        sh:class ex:LeaveRequest ;
    ] ;
    sh:property [
        sh:path ex:performedBy ;
        sh:minCount 1 ;
```

```
        sh:class ex:Employee ;
    ] ;
    sh:property [
        sh:path ex:toStatus ;
        sh:minCount 1 ;
        sh:in ( data:leaveStatus_Pending data:leaveStatus_Approved
                data:leaveStatus_Rejected data:leaveStatus_Escalated ) ;
    ] ;
    sh:property [
        sh:path ex:occurredAt ;
        sh:minCount 1 ;
        sh:datatype xsd:dateTime ;
    ] ;
    sh:property [
        sh:path ex:transitionCode ;
        sh:in ( "Pending->Approved" "Pending->Rejected" "Pending->Escalated"
            "Rejected->Pending" "Escalated->Approved" "Escalated->Rejected" ) ;
        sh:message "transitionCode must be one of the allowed recorded transitions." ;
    ] .
```

How to run validation

```
# From the capstone project folder:
pyshacl -s 04_shapes/04_hrms_capstone_shapes.ttl         02_abox/02_hrms_capstone_abox_broken.ttl         >
05_reports/validation_report_before.txt
```

Why the command can use the ABox here

The sample ABox includes type facts such as data:employee_E001 a ex:Employee. That lets SHACL check class constraints directly. In larger projects you may also pass the TBox/ontology file, but do not add that complexity until this loop is clear.

Expected failed validation report

Focus node	Path	Expected failure
data:leaveRequest_LR101	ex:submittedBy	Missing required Employee link.
data:leaveRequest_LR101	ex:requestedDays	Value "two" is not xsd:integer.
data:leaveRequest_LR102	ex:hasStatus	Draft is not in the allowed status list.
data:task_T101	ex:assignedToRole	Missing required Role link.
data:task_T102	ex:assignedToRole	Value points to an Employee, not a Role.
data:event_EV101	ex:performedBy	Missing required Employee actor.
data:event_EV101	ex:toStatus	Draft is not in the allowed status list.
data:event_EV103	ex:transitionCode	Approved->Pending is not in the allowed transition list.

Fix patch: correct the ABox, not the shape

Replace only the broken blocks below. Do not weaken the SHACL shapes just to make bad data pass.

```
data:leaveRequest_LR101 a ex:LeaveRequest ;
    ex:requestId "LR101" ;
    ex:submittedBy data:employee_E001 ;
    ex:hasStatus data:leaveStatus_Pending ;
    ex:requestedDays 2 .


data:leaveRequest_LR102 a ex:LeaveRequest ;
    ex:requestId "LR102" ;
    ex:submittedBy data:employee_E002 ;
    ex:hasStatus data:leaveStatus_Rejected ;
    ex:requestedDays 2 .


data:task_T101 a ex:LeaveTask ;
    ex:taskId "T101" ;
    ex:taskName "Review LR101" ;
    ex:forRequest data:leaveRequest_LR101 ;
    ex:assignedToRole data:role_HRReviewer ;
    ex:taskStatus data:taskStatus_Open ;
    ex:dueDate "2026-07-09"^^xsd:date .


data:task_T102 a ex:LeaveTask ;
    ex:taskId "T102" ;
    ex:taskName "Review LR102" ;
    ex:forRequest data:leaveRequest_LR102 ;
    ex:assignedToRole data:role_HRReviewer ;
    ex:taskStatus data:taskStatus_Open ;
    ex:dueDate "2026-07-09"^^xsd:date .


data:event_EV101 a ex:LeaveEvent ;
    ex:eventId "EV101" ;
    ex:eventType "Reviewed" ;
    ex:forRequest data:leaveRequest_LR102 ;
    ex:performedBy data:employee_E003 ;
    ex:toStatus data:leaveStatus_Rejected ;
    ex:occurredAt "2026-07-07T10:30:00"^^xsd:dateTime .


data:event_EV103 a ex:LeaveEvent ;
    ex:eventId "EV103" ;
    ex:eventType "Reviewed" ;
    ex:forRequest data:leaveRequest_LR100 ;
    ex:performedBy data:employee_E003 ;
    ex:fromStatus data:leaveStatus_Pending ;
    ex:toStatus data:leaveStatus_Approved ;
    ex:transitionCode "Pending->Approved" ;
    ex:occurredAt "2026-07-07T11:15:00"^^xsd:dateTime .
```

Rerun validation

```
pyshacl -s 04_shapes/04_hrms_capstone_shapes.ttl         02_abox/02_hrms_capstone_abox_fixed.ttl         >
05_reports/validation_report_after.txt

Expected result: Conforms: True
```

A passing report means the data passed these checks. It does not mean the full business process is complete.

Governance evidence

The capstone is not finished when the graph passes. A future reader must know what failed, what changed, why it changed, and who accepted it.

Validation notes example

Item	What to record
Before validation	Conforms: False; record focus nodes and failed paths.
Fix decision	Correct ABox blocks for LR101, LR102, T101, T102, EV101, and EV103.
Reason	The SHACL rules are still correct; the sample data was broken.
After validation	Conforms: True after fixing the ABox.
Files changed	02_hrms_capstone_abox_fixed.ttl and validation reports.

AI Review Log example

Date	Prompt summary	AI suggestion	Human verification	Decision
2026-07-07	Asked AI to review task assignment pattern.	Suggested assigning tasks directly to Nisha.	Rejected: capstone rule assigns tasks to Role first.	Rejected.
2026-07-07	Asked AI to review failed SHACL report.	Listed missing submittedBy, bad days, bad status, missing role, wrong role target, and missing actor.	Matched every item against ABox and shape.	Accepted.
2026-07-07	Asked AI for fix patch.	Proposed changing the shape to allow Draft.	Rejected: Draft is not an approved status.	Rejected and corrected.

Version note example

Field	Example
Version	v1.0 capstone rehearsal package
Changed files	02_hrms_capstone_abox_fixed.ttl, validation_report_after.txt
Reason	Corrected sample records without weakening business checks.
Breaking change?	No. The model and shapes stayed stable; only broken sample data changed.
Validation evidence	validation_report_after.txt shows Conforms: True.
Owner approval	HR data owner accepts allowed statuses and role-based task assignment.

Owner checklist

Checklist

[ ] Departments, employees, roles, requests, tasks, and events have clear business meaning.
[ ] Tasks are assigned to roles, not casually to persons.
[ ] Leave statuses use controlled nodes, not free text.
[ ] SPARQL queries return the expected rows on the broken and fixed data.
[ ] SHACL fails broken data and passes fixed data.
[ ] The AI Review Log shows accepted and rejected suggestions.
[ ] The change log explains why the fix was made.

Governance rule

Chat history is not audit evidence. A future maintainer needs files inside the project folder, not a memory of what someone once discussed.

Final explanation the reader must be able to give

Say it in plain English

The TBox names the HRMS concepts and relationships. The ABox gives sample company, employee, role, leave-request, task, and event facts. SPARQL asks what is visible in the graph. SHACL checks whether the graph is acceptable. The broken ABox intentionally fails because some requests, tasks, and events are incomplete or use invalid values. The fix changes the data, not the rule. A transition check on LeaveEvent also rejects an out-of-order status change, such as jumping back from Approved to Pending, without SHACL ever deciding which transition should happen. Governance notes explain what changed and why.

Capstone pass condition

You are ready to merge the book only if you can do the following without reading the TTL aloud line by line:

Capability	Pass condition
Explain the model	You can describe people, departments, roles, requests, tasks, events, and statuses.
Read the TBox	You can separate reusable model terms from sample facts.
Read the ABox	You can find the broken records by eye before running SHACL.
Run SPARQL	You can explain what the query shows and what it does not prove.
Run SHACL	You can read focus node, path, and failed constraint.
Fix correctly	You fix the ABox without weakening the shape.
Govern	You record validation evidence, AI review, and owner approval.

Four Jobs wall chart - final return

Job	Capstone example	Wrong use to avoid
Model	TBox names LeaveRequest, Role, LeaveTask, and properties.	Do not put workflow actions inside comments.
Ask	SPARQL finds pending requests and task assignments.	Do not treat query rows as validation reports.
Check	SHACL rejects missing submittedBy and wrong task assignment.	Do not use SHACL to send emails or approve requests.
Act	Application code would route approval and notifications later.	Do not hide app behavior in ontology files.
Govern	Change log and AI Review Log record evidence.	Do not rely on memory or chat history.

Outline compliance and self-audit before calling Part 8 ready

Possible issue checked	Verification	Decision
Too much new theory	Only earlier ideas are reused: company hierarchy, leave lifecycle, task/event, SPARQL, SHACL, and governance.	Correct.
Outline output incomplete	Capstone includes Organization, Department, Employee, Role, Task, Event, Status, WorkflowStep transition (fromStatus/transitionCode on LeaveEvent), SPARQL, SHACL, AI Review Log, and governance checklist.	Corrected - added the missing transition check.
TTL/SHACL syntax defects	Corrected missing spaces in LR100, T100, EV100, LeaveTaskShape, and LeaveEventShape. Corrected files parse locally.	Corrected.
App logic hidden in ontology	Approval routing and notification remain application responsibilities.	Boundary preserved.
Queries mistaken for validation	SPARQL sections show visible facts; SHACL section rejects invalid data.	Correct.
Shape weakened to pass data	Fix patch changes ABox only; allowed statuses and role target rules stay strict.	Correct.
Role/person confusion	Tasks use assignedToRole. SHACL rejects an Employee node as a role target.	Correct.
Beginner overload	One capstone package repeats the known loop: see, query, validate, fix, record.	Acceptable for capstone.

Appendices A-F - Reference, Repair, and Lookup Pack

Boundary
These appendices are not hidden chapters. If the reader needs the lesson, they return to the chapter. If they need a command, symptom table, template, or short reminder, they use the appendix.

Appendix	Reader uses it for	Not repeated from chapters
A - Protégé Download and Python Validation Reference	Install or verify tools after the book has introduced them.	No modelling lesson.
B - Error Fixes	Find the smallest likely repair from a symptom.	No full syntax lesson.
C - AI Prompt Pack	Ask AI safely and record review decisions.	No repeated AI examples from chapters.
D - Protégé Visual Labs	Remember which tab shows which concept.	No full walkthroughs.
E - Mini TTL File Bank	Run a tiny known-good smoke test.	Not a second capstone.
F - Glossary	Resolve term confusion quickly.	No theory essay.

Appendix A - Protégé Download and Python Validation Reference

Use this only when setting up or verifying the final workspace. The opening section intentionally installs only Protégé; this appendix is the later full-tool reference.

Tool	Source	Check
Protégé Desktop	https://protege.stanford.edu/	Application opens and can create a new ontology.
VS Code	https://code.visualstudio.com/download	Project folder opens as a folder, not one loose file.
Python	https://www.python.org/downloads/	python --version or py --version returns a version.
pySHACL	https://pypi.org/project/pyshacl/	pyshacl --version returns a version inside the virtual environment.

Windows setup commands

```
# Open Command Prompt inside the project folder.
py -m venv .venv
.venv\Scripts\activate
python -m pip install --upgrade pip
pip install rdflib pyshacl
pyshacl --version
```

Mac / Linux setup commands

```
# Open Terminal inside the project folder.
python3 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
pip install rdflib pyshacl
pyshacl --version
```

[ ] Protégé opens.
[ ] VS Code opens the project folder.
[ ] Python version command works.
[ ] pySHACL version command works after activation.
[ ] The standard six folders exist: 01_tbox, 02_abox, 03_queries, 04_shapes, 05_reports, 06_governance.

Appendix B - Turtle, SPARQL, and SHACL Error Fixes

Start with the symptom. Use the smallest repair that matches the symptom. Do not rewrite the model unless the chapter rule is actually wrong.

B1 - Turtle repair table

Symptom	Likely cause	Smallest repair
Parser expects dot or end of statement.	The subject block was not closed.	Check the last predicate-object line and end it with a dot.
Parser points near a semicolon or comma.	Continuation punctuation was mixed.	Semicolon = new predicate for same subject. Comma = another object for same predicate.
Prefix not found.	A used prefix was not declared.	Add the missing @prefix line before any triples.
Date or integer behaves like text.	Literal was quoted or typed incorrectly.	Use a typed date such as "2026-07-07"^^xsd:date and an unquoted integer such as 3.
The label reads correctly but links are wrong.	Label was treated as identity.	Use stable IRIs for identity; keep labels as human text.

B2 - SPARQL repair table

Symptom	Likely cause	Smallest repair
No rows returned.	Pattern is too strict, wrong IRI, or node/text mismatch.	Comment out one pattern line at a time and compare each IRI to the ABox.
Blank value appears.	OPTIONAL kept the row while the optional link was missing.	Treat the blank as evidence; decide later with SHACL if it is invalid.
Variable is unbound or missing from result.	SELECT variable and WHERE variable do not match.	Use identical variable spelling everywhere.
Broken records still appear.	SPARQL is asking, not validating.	Run SHACL when you need accept/reject evidence.

B3 - SHACL repair table

Report clue	Meaning	Correct action
Conforms: False	At least one checked condition failed.	Read focus node, result path, and message before editing.
Focus node	The exact node that failed.	Open that ABox block first.
Result path	The property that failed.	Check whether it is missing, wrong type, wrong class, or not allowed.
sh:minCount failed	A required value is missing.	Add the missing value only if the requirement is correct.
sh:datatype failed	A literal exists but has the wrong datatype.	Fix the literal; do not weaken the datatype rule.
sh:class failed	A link points to the wrong kind of node.	Change the linked node to the correct class of thing.

Hard rule

Fix bad data in the ABox. Change a shape only when the requirement itself is wrong, and record why.

Appendix C - AI Prompt Pack

Use these prompts as controlled requests. The AI answer is never the evidence; the file, query result, validation report, or owner decision is the evidence.

Use case	Prompt template	Human verification
Review a TBox	Review this TBox only for missing labels, unclear comments, duplicate terms, and class/property confusion. Do not invent new business scope.	Check every suggested term against the outline and current files.
Check Turtle syntax	Inspect this TTL only for prefixes, dots, semicolons, commas, datatypes, and unresolved IRIs. Return issues and minimal fixes.	Parse the file locally.
Predict SPARQL rows	Given this ABox and query, list expected rows and explain why each row appears or does not appear.	Run the query and compare row by row.
Read SHACL report	Explain each failure using focus node, result path, constraint, and likely ABox fix. Do not weaken the shape unless the rule is logically wrong.	Match every failure to the actual ABox block.
Governance review	Convert this reviewed suggestion into an AI Review Log entry with accepted, rejected, or modified decision.	Confirm the decision is traceable in CHANGELOG or report files.

AI Review Log template

Date	Prompt summary	AI suggestion	Human verification	Decision
YYYY-MM-DD	What you asked AI to inspect.	What AI proposed.	What file, query result, validation report, or owner decision you checked.	Accepted / Rejected / Modified

Bad AI habit to block

Do not accept any suggestion that makes bad data pass by weakening a correct rule.

Appendix D - Protégé Visual Labs Reference

This is a memory index for the visual labs, not a second walkthrough. Use it to return to the right tab and then go back to the chapter if you need the lesson.

Concept	Where to look in Protégé	Quick reminder
Triple / object link	Individuals tab + Object Properties	One individual connected to another individual.
Class hierarchy	Classes tab	Use the all-child-is-parent test before creating subclass links.
Inverse links	Object Properties tab	One direction can imply the reverse direction when inverse meaning is declared.
Data properties	Data Properties tab	Use for text, numbers, dates, and other literal values.
TBox vs ABox	Classes/properties vs Individuals	Reusable vocabulary is not the same as sample facts.
Annotations	Annotation panel	Labels and comments help humans; they are not executable rules.
Reasoner view	Reasoner menu + inferred type view	Inferred facts are produced by model meaning, not manually typed.

Protégé boundary

Protégé is the visual surface. SHACL is still the accept/reject proof for data quality.

[ ] I can find classes, object properties, data properties, annotations, and individuals in Protégé.
[ ] I can return from the visual idea to the Turtle mirror without rereading a full chapter.
[ ] I do not treat a comment or label as a rule.

Appendix E - Mini TTL File Bank

These are smoke-test files. They intentionally stay smaller than the capstone. Use them to check that the toolchain works before debugging a larger package.

E1 - 01_tiny_tbox.ttl

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .


ex:Employee a owl:Class ;
    rdfs:label "Employee" ;
    rdfs:comment "A person in the HRMS graph." .


ex:Department a owl:Class ;
    rdfs:label "Department" ;
    rdfs:comment "A business department." .


ex:memberOfDepartment a owl:ObjectProperty ;
    rdfs:label "member of department" ;
    rdfs:comment "Connects an employee to a department." .


ex:employeeId a owl:DatatypeProperty ;
    rdfs:label "employee ID" ;
    rdfs:comment "Stable employee identifier." .
```

E2 - 02_tiny_abox.ttl

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix data: <https://example.org/hrms/data/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .


 data:dept_HR a ex:Department ;
    rdfs:label "Human Resources" .


 data:employee_E001 a ex:Employee ;
    ex:employeeId "E001" ;
    rdfs:label "Ananya Rao" ;
    ex:memberOfDepartment data:dept_HR .
```

E3 - 03_tiny_query.sparql

```
PREFIX ex: <https://example.org/hrms/ontology/>
PREFIX data: <https://example.org/hrms/data/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>


SELECT ?employee ?name
WHERE {
    ?employee a ex:Employee ;
        rdfs:label ?name ;
        ex:memberOfDepartment data:dept_HR .
}
ORDER BY ?employee
```

E4 - 04_tiny_shape.ttl

```
@prefix ex: <https://example.org/hrms/ontology/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .


ex:EmployeeShape a sh:NodeShape ;
    sh:targetClass ex:Employee ;
    sh:property [
        sh:path ex:employeeId ;
        sh:minCount 1 ;
        sh:datatype xsd:string ;
        sh:message "Each employee must have one employee ID." ;
    ] ;
    sh:property [
        sh:path ex:memberOfDepartment ;
        sh:minCount 1 ;
        sh:class ex:Department ;
        sh:message "Each employee must belong to a Department." ;
    ] .
```

E5 - Run order

```
# Validate the tiny sample.
pyshacl -s 04_tiny_shape.ttl 02_tiny_abox.ttl


# Expected validation result:
# Conforms: True
```

File bank rule

If the tiny file passes but the larger project fails, the toolchain is fine. Debug the larger project, not Python or pySHACL.

Appendix F - Glossary and Plain-Language Reference

Use this as a lookup table only. For a full explanation, return to the chapter where the term was first taught.

Term	Plain meaning	Do not confuse with
RDF	Graph model: facts as subject-predicate-object triples.	Turtle syntax.
Turtle / TTL	Readable text syntax for RDF.	The graph model itself.
IRI	Stable identifier for a thing or term.	Human label.
Prefix	Shortcut for a long IRI base.	A created node.
Triple	One subject-predicate-object fact.	A whole workflow.
Class	Reusable type such as Employee.	One specific employee.
Individual	One specific node such as data:employee_E001.	Reusable model vocabulary.
TBox	Reusable classes and properties.	Sample business facts.
ABox	Facts about individuals.	Validation rules.
Object property	Relationship from node to node.	Text or number value.
Data property	Value attached to a node.	Linked business node.
Annotation	Human-readable label/comment.	Executable rule.
OWL / RDFS	Meaning used for hierarchy and inference.	Application workflow.
Reasoner	Tool that derives inferred facts.	Business validator.
Open World Assumption	Missing means unknown, not false.	Closed-world validation.
SPARQL	Asks what the graph says.	Validation.
OPTIONAL	Keeps a row when an optional link is missing.	Proof that data is valid.
FILTER	Restricts query result rows.	Permanent rule.
SHACL	Checks whether data is acceptable.	Application action.
NodeShape	SHACL checker aimed at nodes.	OWL class definition.
sh:targetClass	Which class of nodes a shape checks.	Class creation.
sh:minCount	Minimum required values.	Automatic value creation.
sh:datatype	Expected literal datatype.	Automatic conversion.
Focus node	Node that failed validation.	Random nearby record.
Result path	Property that failed validation.	The full solution.
Application logic	Code that acts: routing, emails, screens.	Ontology file.
Governance	Evidence of decisions and validation.	Chat memory.
AI Review Log	Record of AI suggestion and human decision.	Automatic approval.
Fact reading	Plain-English reading of a property (ex:factReading).	Executable rule or inference.

Final appendix pass condition

Capability	Pass condition
Set up tools	You can verify Protégé, VS Code, Python, and pySHACL.
Repair syntax	You can use the symptom tables without rereading the teaching chapters.
Use AI safely	You can accept, reject, or modify AI suggestions with evidence.
Repeat visual labs	You can find the right Protégé tab quickly.
Use file bank	You can run the tiny files before debugging a larger package.
Use glossary	You can resolve term confusion without mixing model, ask, check, act, and govern.

Appendix self-audit before release

Possible issue checked	Verification	Decision
Repeated chapter teaching	Removed full walkthrough style. Kept only commands, lookup tables, templates, tiny smoke-test files, and glossary rows.	Corrected.
Duplicate AI example from capstone	Removed the Nisha/task-assignment example and kept a blank AI Review Log template only.	Corrected.
Part 7 folder/workflow repetition	Kept only setup checklist; detailed workflow remains in Part 7 and capstone.	Corrected.
Turtle/SPARQL/SHACL examples too large	Repair appendix is symptom-based; code examples are only in the mini file bank.	Correct.
Hidden new theory after capstone	No new concepts introduced; every appendix item is reference or repair support.	Correct.
Mini file bank scope creep	Mini files are smaller than capstone and labelled smoke tests.	Correct.