Table of Contents

1 101 – Mein erster Bericht 1
1.1 01 Bino BI installieren 1
1.1.1 01.01 macOS / Linux 1
1.1.2 01.02 Windows (PowerShell) 1
1.1.3 01.03 Einrichtung abschließen 1
1.1.4 01.04 Visual Studio Code und die Bino Reports Extension 2
1.1.4 a) 01.04.01 Was die Extension bietet 2
1.1.4 b) 01.04.02 Konfiguration 3
1.2 02 Projekt initialisieren 3
1.2.1 02.01 Ein neues Projekt anlegen 3
1.2.2 02.02 Die Projektstruktur verstehen 4
1.2.3 02.03 Die bino.toml 5
1.3 03 Eine DataSource anlegen 5
1.3.1 03.01 Die CSV-Datei bereitstellen 6
1.3.2 03.02 Das DataSource-Manifest mit dem Wizard erstellen 6
1.3.3 03.03 Das erzeugte Manifest 7
1.4 04 Ein DataSet anlegen 7
1.4.1 04.01 Das DataSet mit dem Wizard erstellen 8
1.4.2 04.02 Bino Report VSCode Extension - DataSet Preview 8
1.4.3 04.03 query – die SQL-Abfrage im Detail 9
1.4.4 04.04 dependencies – die Abhängigkeiten 9
1.4.5 04.05 Was passiert unter der Haube? 9
1.5 05 Eine Berichtsseite erstellen 9
1.5.1 05.01. Das LayoutPage-Manifest anlegen 10
1.5.1 a) 05.01.01 Was bedeuten die einzelnen Felder? 11
1.5.2 05.02 Den Seitenkopf konfigurieren 11
1.5.2 a) 05.02.01 Was bedeuten die Titel-Felder? 12
1.5.3 05.03 Die Tabelle konfigurieren 13
1.5.3 a) 05.04 Was bedeuten die Tabellen-Felder? 15
1.6 06 Das ReportArtefact anlegen 16

1.7 07 Vorschau im Browser 17
1.8 08 Fehler finden mit dem Linter 18
1.8.1 08.01 Manifest-Struktur prüfen 18
1.8.2 08.02 Auch die Daten prüfen 18
1.9 09 Das PDF erzeugen 19
1.10 10 Zusammenfassung 19
2 104 - Die Vorschau mit bino preview 21
2.1 01 Was macht bino preview? 21
2.2 02 Grundlegende Verwendung 22
2.2.1 02.01 Vorschau starten 22
2.2.2 02.02 Vorschau beenden 23
2.2.3 02.03 Anderes Projektverzeichnis 23
2.3 03 Flags und Optionen 23
2.3.1 03.01 Übersicht aller Flags 23
2.3.2 03.02 Port ändern 23
2.3.3 03.03 Mehrere Flags kombinieren 24
2.4 04 Hot-Reload: Automatisches Neuladen 24
2.4.1 04.01 Welche Dateien werden überwacht? 24
2.4.2 04.02 Workflow: Bearbeiten, Speichern, Sehen 24
2.5 05 SQL-Logging mit –log-sql 25
2.5.1 05.01 SQL-Logging aktivieren 25
2.6 06 Lint-Regeln in der Vorschau 25
2.6.1 06.01 Was bewirkt –lint? 25
2.6.2 06.02 Vergleich: Lint in preview vs. build 25
2.7 07 Datenvalidierung 26
2.7.1 07.01 Validierungsmodi 26
2.7.2 07.02 Was wird geprüft? 27
2.7.3 07.03 Stichprobengröße anpassen 27
2.8 08 Konfiguration über bino.toml 27
2.8.1 08.01 Standard-Flags setzen 27
2.8.2 08.02 Kommandozeile überschreibt bino.toml 28

2.8.3 08.03 Umgebungsvariablen setzen 28

2.9 09 Hooks: Automatisierung in der Vorschau 28

2.9.1 09.01 Verfügbare Checkpoints für die Vorschau 29

2.9.2 09.02 Beispiel: Testdaten vor der Vorschau generieren 29

2.9.3 09.03 Beispiel: Daten vor jedem Refresh aktualisieren 29

2.9.4 09.04 Fehlerbehandlung bei Hooks 29

2.9.5 09.05 Umgebungsvariablen in Hooks 30

2.9.6 09.06 Exit Code 78: Hook überspringen 30

2.10 10 Constraints mit mode==preview 30

2.11 11 Umgebungsvariablen in Manifesten 31

2.11.1 11.01 Syntax 31

2.11.2 11.02 Verhalten in preview vs. build 31

2.12 12 Unterschiede: preview vs. build vs. serve 32

2.12.1 12.01 Vergleich auf einen Blick 32

2.12.2 12.02 Wann welchen Befehl verwenden? 32

2.13 13 Praxistipps für die Vorschau 33

2.13.1 13.01 Schnellere Iteration mit Sampling 33

2.13.2 13.02 SQL debuggen mit –log-sql 33

2.13.3 13.03 Lint aktivieren für frühes Feedback 33

2.13.4 13.04 Empfohlene bino.toml für die Entwicklung 34

2.13.5 13.05 VS Code und bino preview 34

2.14 14 Die Benutzeroberfläche der Vorschau 34

2.14.1 14.01 Übersicht 35

2.14.2 14.02 Artefakt-Dropdown 35

2.14.3 14.03 Assets Modal 36

2.14.4 14.04 Dependency-Graph 38

2.14.5 14.04 Data Explorer 38

2.14.6 14.05 Suchleiste 40

2.14.7 14.06 Warning-Panel 40

2.14.8 14.07 Present Mode 41

2.14.8 a) 14.07.01 Beispiel 44

2.15 15 Zusammenfassung 44

3 103 - DataSource und DataSet 46
3.1 01 Das Datenmodell im Überblick 46
3.2 02 DataSource: Struktur und Typen 47
3.2.1 02.01 Grundstruktur einer DataSource 47
3.2.2 02.02 Namensregeln 49
3.2.3 02.03 Unterstützte Typen (spec.type) 49
3.2.4 02.04 Beispiel: Inline-DataSource 50
3.2.5 02.05 Beispiel: CSV-DataSource 50
3.2.5 a) CSV-Reader-Optionen im Überblick 51
3.2.6 02.06 Beispiel: Excel-DataSource 51
3.2.7 02.07 Beispiel: Parquet-DataSource 52
3.2.8 02.08 Beispiel: PostgreSQL-DataSource 52
3.2.9 02.09 Caching mit ephemeral 53
3.2.10 02.10 Sampling – große Datenmengen während der Entwicklung begrenzen 53
3.2.11 02.11 DataSources erweitern mit Plugins 54
3.2.12 02.12 Attribut-Referenz DataSource 54
3.3 03 DataSet: Daten transformieren mit SQL 56
3.3.1 03.01 Grundstruktur eines DataSets 56
3.3.2 03.02 Tabellenname in der Abfrage 57
3.4 04 query - die SQL-Abfrage im Detail 57
3.4.1 04.01 Unterstützter SQL-Umfang 57
3.4.2 04.02 Pass-Through vs. eigene Query 58
3.4.3 04.03 Externe SQL-Dateien 59
3.4.4 04.04 PRQL als Alternative zu SQL 60
3.5 05 dependencies - die Abhängigkeiten 60
3.5.1 05.01 Ausführungsreihenfolge sicherstellen 61
3.5.2 05.02 Tabellen in der Query-Engine registrieren 61
3.5.3 05.03 Beispiel mit mehreren Abhängigkeiten (Join) 61
3.5.4 05.04 Wann kann dependencies weggelassen werden? 62
3.6 06 DataSet-Verkettung: DataSets auf DataSets 62
3.7 07 Inline-DataSets und Inline-DataSources 63

3.7.1 07.01 Inline-DataSet in einer Komponente 63
3.7.2 07.02 Inline-DataSet mit Inline-DataSource 64
3.7.3 07.03 Benannte und Inline-Referenzen mischen 64
3.8 08 Das Standard-Datenschema 65
3.8.1 08.01 Messwerte (Szenarien) 65
3.8.2 08.02 Gruppierungsdimensionen 65
3.8.3 08.03 Metadaten- und Steuerfelder 66
3.8.4 08.04 Beispielabfrage für in standardkonformes DataSet 66
3.9 09 Datenvalidierung 66
3.9.1 09.01 Was wird geprüft? 67
3.9.2 09.02 Validierung konfigurieren 67
3.9.3 09.03 Stichprobengröße 68
3.10 10 Constraints: DataSets bedingt einbinden 68
3.11 11 Vollständiges Beispiel: Von der Quelle zum Chart 68
3.12 12 Zusammenfassung 70
4 104 – Seitenlayout und LayoutCards 72
4.1 01 – Die LayoutPage im Überblick 72
4.1.1 01.01 Grundgerüst einer LayoutPage 73
4.2 02 – Seitenformat und Ausrichtung 73
4.2.1 02.01 pageFormat – das Seitenformat 73
4.2.2 02.02 pageOrientation – die Ausrichtung 74
4.3 03 – Der Seitenkopf 74
4.3.1 03.01. Alle Titelfelder im Überblick 74
4.3.2 03.02 Beispiel: Vollständig konfigurierter Seitenkopf 75
4.3.3 03.03 titleDateLink – Start- und Enddatum verknüpfen 76
4.4 04 – Der Nachrichtenbereich (Action Title) 77
4.4.1 04.01 Beispiel 77
4.5 05 – Die Fußzeile 77
4.6 06 – Das Seitenlayout (pageLayout)/ Das Cardlayout (cardLayout) 78
4.6.1 06.01 Verfügbare Layouts 78
4.6.2 06.02 Visueller Überblick 79

4.6.3 06.03 cardCustomTemplate 79
4.6.4 06.04 Beispiel: split-vertical – zwei Spalten 80
4.6.5 06.05 Beispiel: 2x2 – vier Quadranten 80
4.6.6 06.06 Beispiel: custom-template 81
4.7 07 – LayoutCards – Inhaltsblöcke mit eigenem Titel 82
4.7.1 07.01 Grundstruktur 82
4.7.2 07.02 Verfügbare Felder einer LayoutCard 83
4.7.3 07.03 Beispiel: Zwei LayoutCards nebeneinander 83
4.7.4 07.04 LayoutCard vs. direkte Komponenten 85
4.8 08 – Verfügbare Kind-Komponenten 85
4.9 09 – Parameter und dynamische Seiten 86
4.9.1 09.01 Parameter deklarieren 86
4.9.2 09.02 Parametertypen 87
4.9.3 09.03 Parameter verwenden 87
4.9.4 09.04 Parameter beim Einbinden übergeben 87
4.10 10 – Alles zusammen: ein vollständiges Dashboard 88
4.11 11 Zusammenfassung 90
5 105 - Constraints: Manifeste bedingt einbinden 91
5.1 01 Was sind Constraints? 91
5.1.1 01.01 Constraints in einer Table-Komponente 92
5.2 02 Syntax und Ausdrücke 93
5.2.1 02.01 Unterstützte Operatoren 93
5.2.2 02.02 Verfügbare Felder 93
5.2.3 02.03 Strukturierte Schreibweise 94
5.2.4 02.04 Mehrere Bedingungen (UND-Verknüpfung) 94
5.3 03 Modus-Constraints: preview, build, serve 94
5.3.1 03.01 Nur in der Vorschau (mode==preview) 95
5.3.2 03.02 Nur im Build (mode==build) 95
5.3.3 03.03 Alles außer Build (mode!=build) 95
5.4 04 Sampling mit Constraints steuern 96
5.5 05 Format-Constraints: Ausgabeformate unterscheiden 97

5.5.1 05.01 Beispiel: Detailliert für Druck, kompakt für Bildschirm 97

5.6 06 Welche Manifest-Typen unterstützen Constraints? 98

5.7 07 Best Practices 99

5.7.1 07.01 Gleicher Name, unterschiedliche Constraints (Scoped Names) 99

5.7.2 07.02 Constraints sparsam einsetzen 100

5.7.3 07.03 Constraints dokumentieren 100

5.8 08 Zusammenfassung 100

6 106 - Assets und Bilder 102

6.1 01 Das Asset-Manifest 102

6.1.1 01.01 Die drei Quellen 103

6.2 02 Assets referenzieren: zwei Schreibweisen 103

6.3 03 Lokale Bilddateien 104

6.4 04 Remote-Bilder und Inline-Base64 104

6.4.1 04.01 remoteURL 104

6.4.2 04.02 inlineBase64 105

6.5 05 Assets als eigenständige, wiederverwendbare Komponenten 105

6.6 06 Beispiel: Alle drei Referenz-Arten in einer Seite 106

6.7 07 Zusammenfassung 107

7 107 - Grid 109

7.1 01 Minimalkonfiguration 109

7.2 02 Darstellung des Rasters 110

7.3 03 Kind-Elemente platzieren 110

7.4 04 Grid vs. custom-template 111

7.5 05 Beispiel: Tabelle und Chart im Vergleich 111

7.6 06 Zusammenfassung 113

8 108 - Bino-Werkzeuge für den Alltag 114

8.1 01 Cache-Verwaltung: bino cache clean 114

8.2 02 Abhängigkeiten installieren: bino setup 115

8.3 03 Bino selbst aktualisieren: bino update 115

8.4 04 Shell-Autovervollständigung: bino completion 116

8.5 05 Projekt- und Versionsinformationen 116

8.6 06 Projekte aus Vorlagen starten: bino init 116
8.6.1 06.01 Sicherheit bei entfernten Vorlagen 117
8.7 07 Editor- und KI-Integration: bino lsp und bino mcp 117
8.7.1 07.01 bino lsp 118
8.7.2 07.02 bino mcp 118
8.7.3 07.03 Der gemeinsame Daemon 118
8.8 08 Zusammenfassung 100
9 109 – Eine Seite, viele Berichte: LayoutPage-Parameter 120
9.1 01 Das Problem an einem Beispiel 120
9.2 02 Die parametrisierte Seite 121
9.3 03 Die Seite mehrfach einbinden 122
9.4 04 Die Parametertypen 123
9.5 05 Rangfolge: Was gewinnt? 124
9.6 06 Parameter auf referenzierten Komponenten 124
9.7 07 Parameter in bino preview 125
9.8 08 Wann Parameter – und wann nicht 126
9.9 09 Typische Fehler 126
9.10 10 Tipps 127
10 201 – Fortgeschrittener Bericht: Mehrseiter mit Tree, Drill-Down und Live-Artefakt 128
10.1 01 Überblick: Was wir bauen 128
10.2 02 Die DataSets anlegen 129
10.2.1 02.01 DataSet 1: bb_example_dataset – Pass-Through 129
10.2.2 02.02 DataSet 2: bb_example_dataset_4_bn201 – Hierarchie umstrukturieren 130
10.2.2 a) Was passiert in der SQL-Abfrage? 133
10.2.3 02.03 DataSet 3: bb_example_dataset_4_bn201_regions – Weltweite Aggregation 133
10.2.4 02.04 Warum drei DataSets? 136
10.3 03 Seite 1: Tabelle, Text und Zeitreihe mit Custom-Template 136
10.3.1 03.01 Custom-Template-Layout definieren 136
10.3.2 03.02 Das vollständige LayoutPage-Manifest 137
10.3.3 03.03 Neue Konzepte auf dieser Seite 139
10.3.3 a) Die Text-Komponente 139

10.3.3 b) ChartTime mit gestapelten Balken (stack) 140
10.3.3 c) Gruppierte Tabelle (grouped: true) 140
10.3.3 d) Mehrere Varianzen und Bar-Spalten 140
10.4 04 Seite 2: Die Tree-Komponente (Treiberanalyse) 141
10.4.1 04.01 Constraints auf Seitenebene 141
10.4.2 04.03 Die Tree-Komponente im Detail 142
10.4.3 04.04 Tree-Attribute im Überblick 146
10.4.3 a) Globale Tree-Attribute 146
10.4.3 b) Edge-Definition 146
10.4.3 c) Node-Definition 147
10.4.4 04.05 Wie ref und filter zusammenspielen 147
10.5 05 Seiten 3 und 4: Parametrisierte Drill-Down-Seite 148
10.5.1 05.01 Parameter deklarieren 148
10.5.2 05.02 Parameter in Texten und Filtern verwenden 148
10.5.3 05.03 Das vollständige LayoutPage-Manifest 149
10.5.4 05.04 Was auf dieser Seite passiert 152
10.6 06 ReportArtefact und LiveReportArtefact 153
10.6.1 06.01 Parametrisierte Seiten einbinden 155
10.6.2 06.02 Funktion des LiveReportArtefact 155
10.7 07 Vorschau und Build 155
10.7.1 07.01 Vorschau starten 155
10.7.2 07.02 Live-Anwendung starten 156
10.7.3 07.03 PDF erzeugen 156
10.8 08 Zusammenfassung 100
11 202 – Die Table-Komponente 158
11.1 01 Minimalkonfiguration 109
11.2 02 Attribute im Überblick 159
11.2.1 02.01 Datenanbindung, Hierarchie und Sortierung 159
11.2.2 02.02 Szenarien und Varianzen 160
11.2.3 02.03 Maßeinheiten und Skalierung 160
11.2.4 02.04 Formatierung 161

11.2.5 02.05 Darstellung und Layout 161
11.2.6 02.06 Drill-Down 162
11.2.7 02.07 Internationalisierung 163
11.2.8 02.08 Vererbte Werte (Inherited Props) 163
11.3 03 Varianz-Namensschema 164
11.4 04 Inline-Balkendiagramme 165
11.5 05 Beispiel: Gruppierte Tabelle mit Inline-Balken 166
11.5.1 05.01 Was hier passiert 167
11.6 06 Beispiel: Gefilterte Detail-Tabelle 168
11.6.1 06.01 Filter und Parameter 169
11.7 07 Proportionale Skalierung (scale) 169
11.8 08 Drill-Down Beispiel: thereof, partof, columnthereof 170
11.9 09 Table als wiederverwendbare Component 171
11.10 06 Tipps 172
11.11 07 Zusammenfassung 107
12 203 – Die ChartStructure-Komponente 174
12.1 01 Minimalkonfiguration 109
12.2 02 Attribute im Überblick 159
12.2.1 02.01 Datenanbindung, Hierarchie und Sortierung 159
12.2.2 02.02 Szenarien und Varianzen 160
12.2.3 02.03 Maßeinheiten und Skalierung 160
12.2.4 02.04 Darstellung 176
12.2.5 02.05 Internationalisierung 177
12.3 03 Gestapelte Balken mit stack 177
12.3.1 03.01 Beispiel: Gestapelte Balken 178
12.4 04 Vererbte Werte (Inherited Props) 179
12.5 05 Overdrive-Balken bei Überlauf 180
12.6 06 Beispiel: ChartStructure als wiederverwendbare Component 180
12.6.1 06.01 Was hier konfiguriert ist 181
12.7 07 Beispiel: ChartStructure mit Filter im Tree-Knoten 181
12.8 08 ChartStructure vs. Table 181

12.9 09 Tipps 182

13 204 – Die ChartTime-Komponente 183

13.1 01 Minimalkonfiguration 109

13.2 02 Attribute im Überblick 159

13.2.1 02.01 Darstellungsmodus und Zeitachse 184

13.2.2 02.02 Aggregationsebene und Sortierung 184

13.2.3 02.03 Szenarien und Varianzen 185

13.2.4 02.04 Maßeinheiten und Skalierung 185

13.2.5 02.05 Sichtbarkeit und Overlays 186

13.2.6 02.06 Proportionale Skalierung 186

13.2.7 02.07 Internationalisierung 163

13.3 03 Chart-Modi im Detail 187

13.3.1 03.01 Balkenmodus (chartMode: bar) 187

13.3.2 03.02 Linienmodus (chartMode: line) 187

13.3.3 03.03 Automatischer Modus (chartMode: auto) 187

13.4 04 Zeitintervall (dateInterval) 188

13.5 05 Gestapelte Diagramme mit stack 188

13.5.1 05.01 Beispiel: Gestapeltes Balkendiagramm 189

13.5.2 05.02 Beispiel: 100 % gestapelt 190

13.5.3 05.03 Beispiel: Gestapeltes Flächendiagramm 190

13.5.4 05.04 Stapelmodi im Vergleich 191

13.6 06 Overdrive-Balken bei Überlauf 191

13.7 07 Vererbte Werte (Inherited Props) 191

13.8 08 Beispiel: ChartTime mit Filter auf der Drill-Down-Seite 192

13.9 09 ChartTime vs. ChartStructure 193

13.10 10 Tipps 127

14 205 – Die Tree-Komponente 195

14.1 01 Aufbau eines Tree 195

14.2 02 Globale Attribute 196

14.2.1 02.01 Richtung wählen 197

14.3 03 Kanten (edges) 197

14.3.1 03.01 Kanten-Attribute 197

14.3.2 03.02 Kantenstil (edges[].style) 198
14.3.3 03.03 Operatoren 198
14.4 04 Knoten (nodes) 199
14.4.1 04.01 Knoten-Attribute 199
14.5 05 Knotenarten (kind) 200
14.5.1 05.01 Label 200
14.5.2 05.02 Table 200
14.5.3 05.03 ChartStructure 201
14.5.4 05.04 ChartTime 201
14.5.5 05.05 Image 202
14.6 06 DAG-Unterstützung 202
14.7 07 Vollständiges Beispiel: ROI-Treiberanalyse 203
14.8 08 Vollständiges Beispiel: Gemischter Baum mit Charts und Tabellen 204
14.9 09 Wiederverwendbare Components mit ref und Überschreibungen 205
14.10 10 Tipps 127
15 206 – Die Text-Komponente 207
15.1 01 Minimalkonfiguration 109
15.2 02 Attribute im Überblick 159
15.3 03 Unterstützte Markdown-Elemente 208
15.3.1 03.01 Asset-Bilder (asset:) 210
15.4 04 Template-Ausdrücke 210
15.4.1 04.01 Verfügbare Variablen 211
15.4.2 04.02 Beispiele 211
15.4.3 04.03 Sicherheit 212
15.5 05 Skalierung (scale) 212
15.6 06 Beispiel aus dem Beispielbericht 213
15.7 07 Standalone-Manifest 214
15.8 08 Text im Layout platzieren 214
15.9 09 Tipps 182
16 207 - ComponentStyle 217
16.1 01 Grundstruktur 217

16.2 02 global: Typografie und Grundfarben 218
16.3 03 bn-chart-time / bn-chart-structure: Diagramm-Feinsteuerung 218
16.4 04 bn-tree und bn-grid 220
16.5 05 Vollständiges Beispiel 220
16.6 06 Verwandt: ScalingGroup – Skalierung über Komponenten synchronisieren 221
16.7 07 Was dieses Kapitel (noch) nicht beantworten kann 222
16.8 08 Zusammenfassung 100

17 208 - Internationalisierung 224
17.1 01 Das Internationalization-Manifest 224
17.2 02 i18nNamespace: einen Namensraum verknüpfen 225
17.3 03 Übersetzungen ausgeben 226
17.4 04 Beispiel: dasselbe Artefakt in zwei Sprachen 226
17.5 05 Typische Stolpersteine 228
17.6 06 Zusammenfassung 113

18 209 - LiveReportArtefact im Detail 230
18.1 01 Grundstruktur 217
18.2 02 queryParams: Typen im Überblick 231
18.3 03 select: statisch vs. dataset-gestützt 231
18.3.1 03.01 Statische Liste (aus Kapitel 201) 231
18.3.2 03.02 Aus einem DataSet 232
18.4 04 number/number_range: Grenzen und Schrittweite 233
18.5 05 Beispiel: Zwei Routen, ein Parameter, zwei Optionsquellen 233
18.6 06 Zusammenfassung 113

19 210 - Hooks 236
19.1 01 Die neun Checkpoints 236
19.2 02 Umgebungsvariablen 237
19.3 03 Selbst überspringen: Exit-Code 78 238
19.4 04 Beispiel: Jeden Build-Checkpoint protokollieren 238
19.5 05 Typische Anwendungsfälle 239
19.6 06 Zusammenfassung 113

20 211 – Streu- und Blasendiagramme: ChartScatter und ChartBubble 240

20.1 01 Die Datenstruktur: eine Zeile, ein Punkt 240
20.2 02 Das erste Streudiagramm 241
20.2.1 02.01 Kurzform für Achsen 242
20.3 03 Achsen konfigurieren 242
20.4 04 Iso-Linien: die dritte Kennzahl ohne dritte Achse 243
20.5 05 ChartBubble: die dritte Kennzahl als Fläche 244
20.6 06 Tortenanteile und Vergleichsszenario 245
20.7 07 Farben, Beschriftungen und Legende 246
20.8 08 Welche Komponente wann? 246
20.9 09 Tipps 182
21 212 – Ziel-Ist-Vergleiche: die ChartBullet-Komponente 248
21.1 01 Die Datenstruktur 248
21.2 02 Der erste Bullet-Graph 249
21.3 03 Warum Normalisierung der entscheidende Punkt ist 250
21.4 04 Ist und Ziel explizit setzen 250
21.5 05 Attribute im Überblick 251
21.6 06 Der klassische Bullet-Graph 252
21.7 07 IBCS-Modus oder klassisch? 253
21.8 08 Typische Fehler 254
21.9 09 Tipps 182
22 213 – Szenario-Regeln steuern: das RuleSet-Manifest 255
22.1 01 Was ein RuleSet steuert 255
22.2 02 Die eingebauten Standardwerte 256
22.3 03 Die Spaltenreihenfolge ändern 256
22.4 04 _default oder benannt? 257
22.5 05 Vererbung: einmal setzen, überall nutzen 258
22.6 06 colorIndex: die Farbcodierung 259
22.7 07 Die fallback-Regel 260
22.8 08 Die JSON-Schreibweise 261
22.9 09 Wo ruleset verfügbar ist 261
22.10 10 Tipps 127
23 214 – Bausteine teilen: die bino Registry 263

23.1 01 Die beteiligten Dateien 263
23.2 02 Abhängigkeiten deklarieren 264
23.3 03 Der Lockfile-Workflow 264
23.4 04 Installierte Komponenten verwenden 265
23.5 05 Namenskonflikte 266
23.6 06 Registry konfigurieren 267
23.7 07 Authentifizierung 267
23.8 08 Veröffentlichen 268
23.9 09 Was sich gut als Paket eignet 268
23.10 10 Tipps 127

24 301 - Cloud-Datenquellen und Secrets 270
24.1 01 Acht Secret-Typen 270
24.2 02 Zwei grundlegend verschiedene Referenzierungs-Arten 271
24.2.1 02.01 Datenbank-Secrets: explizit über connection.secret 271
24.2.2 02.02 Objektspeicher-Secrets: automatisch über scope 272
24.3 03 Auth-Blöcke im Detail 273
24.3.1 03.01 postgres/mysql 273
24.3.2 03.02 s3 273
24.3.3 03.03 gcs/r2 273
24.3.4 03.04 azure 274
24.3.5 03.05 http (inkl. Proxy) 274
24.3.6 03.06 huggingface 274
24.4 04 provider: credential_chain 274
24.5 05 Zusammenfassung 275

25 302 - DocumentArtefact: Markdown-Dokumente als PDF 276
25.1 01 Grundstruktur 217
25.2 02 Inhaltsverzeichnis und Kapitelnummerierung 277
25.3 03 LaTeX-Mathematik mit KaTeX 277
25.4 04 Kopf- und Fußzeilen 278
25.5 05 Wie dieses Handbuch selbst gebaut wird 278
25.6 06 Ein eigenes, kleineres Beispiel 279
25.7 07 Zusammenfassung 107

26 303 - ScreenshotArtefact 281
26.1 01 Grundstruktur 217
26.2 02 Beispiel aus dem Projekt 282
26.3 03 filenamePattern: ref vs. index 283
26.4 04 Praktische Hinweise 283
26.5 05 Zusammenfassung 275
27 304 - Digitale Signatur von PDFs 285
27.1 01 Ein Test-Zertifikat erzeugen 285
27.2 02 Das SigningProfile-Manifest 285
27.3 03 Ein ReportArtefact signieren 287
27.4 04 Signatur überprüfen 287
27.5 05 Zusammenfassung 275
28 305 - Plugin-Engine 289
28.1 01 Deklaration in bino.toml 289
28.2 02 Binaries: Namenskonvention und Suchpfade 290
28.3 03 Was ein Plugin beisteuern kann 290
28.4 04 Plugins auflisten und ausführen 291
28.5 05 Schema eines Plugin-Kinds einsehen 291
28.6 06 Zusammenfassung 113
29 306 - CI/CD und reproduzierbare Builds 293
29.1 01 Harte Lint-Gates in CI: --fail-on-warnings 293
29.1.1 01.01 Welche Regeln laufen? 294
29.2 02 Abhängigkeiten inspizieren: bino graph 295
29.3 03 Reproduzierbare Builds: SOURCE_DATE_EPOCH 296
29.4 04 --data-mode: eingebettete vs. verlinkte Daten 296
29.5 05 Zusammenfassung 275
30 307 – XY-Charts im Detail: Token, Ebenen und Skalierung 298
30.1 01 Die Grammatik der Measure-Token 298
30.1.1 01.01 Szenario-Slots 298
30.1.2 01.02 Varianz-Token 299
30.1.3 01.03 Die Regel der Szenario-Familie 299
30.2 02 Ebenen: level, seriesLevel und facet 299

30.2.1 02.01 Die Entdopplung – und warum sie stillschweigend passiert 300
30.2.2 02.02 Small Multiples 300
30.3 03 limit: zwei Komponenten, zwei Bedeutungen 301
30.4 04 ScalingGroup: Flächen vergleichbar machen 302
30.4.1 04.01 Map-Scale-Semantik 302
30.4.2 04.02 Warum scale bei Blasen quadratisch wirkt 303
30.5 05 Iso-Linien: Voraussetzungen 303
30.6 06 Welche Komponente liest welche RuleSet-Eigenschaft? 303
30.7 07 Seitenverhältnis und Layout 304
30.8 08 Checkliste für XY-Charts 304
30.9 09 Tipps 182
31 308 – Das Szenario-Modell: RuleSets im Detail 306
31.1 01 Die drei Ebenen 306
31.1.1 01.01 Der Merge in der Praxis 306
31.2 02 Von Präfix zu Variante 308
31.3 03 Varianz-Präfixe 308
31.4 04 Wer liest was? 309
31.5 05 Woher die sichtbaren Kürzel kommen 309
31.6 06 RuleSets debuggen 310
31.7 07 Muster für die Praxis 310
31.7.1 07.01 Hausweite Konvention 310
31.7.2 07.02 Ausnahme für eine einzelne Seite 311
31.7.3 07.03 Umgebungsabhängige Regeln 311
31.8 08 Tipps 312
32 309 – bino im Container: reproduzierbare Builds mit Docker 313
32.1 01 Die zwei Varianten 313
32.2 02 Der erste Lauf 314
32.3 03 Das Projekt mounten 315
32.3.1 03.01 Secrets und Variablen hineinreichen 315
32.4 04 Dateirechte 315
32.4.1 04.01 Ein Shell-Wrapper 316

32.5 05 Preview und Serve im Container 317
32.6 06 Ein neues Projekt anlegen 317
32.7 07 Was offline funktioniert 318
32.8 08 Die Engine-Version im Image 318
32.9 09 Fehlerdiagnose 319
32.10 10 Tipps 127

101 – Mein erster Bericht

> **Disclaimer:** Die in diesem Kapitel vorgestellten Component-Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).

In diesem Kapitel zeigen wir dir Schritt für Schritt, wie du ein Bino-BI-Projekt anlegst und daraus einen einseitigen PDF-Bericht mit Titel und Tabelle erzeugst. Du lernst dabei den vollständigen Arbeitsablauf kennen: Projekt initialisieren, Datenquelle anbinden, Daten aufbereiten, Seite gestalten, Vorschau nutzen und das fertige PDF exportieren.

Wir bleiben bewusst auf einer allgemeinen Ebene – tiefere Erläuterungen zu einzelnen Themen wie zum Beispiel Constraints, Filter oder Parameter folgen in den späteren Kapiteln.

---

01 Bino BI installieren

Bevor du loslegen kannst, muss das Bino-CLI auf deinem System installiert sein. Eine ausführliche Installationsanleitung findest du unter https://cli.bino.bi/getting-started/installation/ (https://cli.bino.bi/getting-started/installation/).

01.01 macOS / Linux

```
/bin/bash -c "$(curl -fsSL https://github.com/bino-bi/bino-cli-releases/releases
```

Alternativ über Homebrew:

```
brew install bino-bi/tap/bino-cli
```

01.02 Windows (PowerShell)

```
irm https://github.com/bino-bi/bino-cli-releases/releases/latest/download/instal
```

01.03 Einrichtung abschließen

Nach der Installation führe einmalig den Setup-Befehl aus. Er lädt die Browser-Runtime herunter, die BinoBI für das PDF-Rendering benötigt:

```
bino setup
```

Prüfe anschließend, ob alles funktioniert:

```
bino version
```

Die Ausgabe sollte die installierte Version anzeigen und mit Exit-Code 0 enden.

01.04 Visual Studio Code und die Bino Reports Extension

Wir empfehlen, Berichte ausschließlich mit **Visual Studio Code** und der **Bino Reports Extension** zu erstellen. Die Extension macht VS Code zum vollwertigen Arbeitsplatz für BinoBI-Projekte – du brauchst kein separates Terminal und kein zusätzliches Tool.

Installiere die Extension über den VS Code Marketplace:

- Bino Reports Extension installieren (https://marketplace.visualstudio.com/items?itemName=BinoBI.vscode-bino)

Alternativ über die Kommandozeile:

```
code --install-extension BinoBI.vscode-bino
```

01.04.01 Was die Extension bietet

Funktion	Beschreibung
Schema-Validierung	Manifest-Dateien werden beim Speichern automatisch gegen das BinoBI-Schema validiert. Fehler erscheinen direkt im Problems-Panel.
Bino Explorer	Alle Dokumente im Projekt werden im Explorer-Seitenbereich nach `kind` gruppiert angezeigt (DataSource, DataSet, LayoutPage etc.).
Autovervollständigung	Kontextbezogene Vorschläge: DataSet-Namen bei `dataset:`, Spaltennamen bei `scenarios:` und `variances:`, gültige Werte bei `kind:` u. v. m.
Go to Definition	Mit `Cmd+Klick` (macOS) bzw. `Strg+Klick` (Windows/Linux) auf Referenzen wie `dataset:` oder `dependencies:` springst du direkt zur Definition.
CLI-Steuerung	Die Befehle `bino build`, `bino preview` und `bino lint` lassen sich direkt aus VS Code starten – über die Kommandopalette oder

Funktion	Beschreibung
	den integrierten Task-Runner.
Lint-Diagnostik	Lint-Ergebnisse werden als Warnungen und Fehler im Problems-Panel angezeigt. So siehst du Probleme sofort, ohne das Terminal zu prüfen.
Spalten-Introspection	Bei `scenarios:` und `variances:` führt die Extension die referenzierte DataSet-Abfrage per DuckDB aus und schlägt die verfügbaren Spaltennamen vor.

01.04.02 Konfiguration

Die Extension funktioniert ohne weitere Konfiguration, solange `bino` im `PATH` verfügbar ist. Bei Bedarf kannst du folgende Einstellungen in den VS Code Settings anpassen:

Einstellung	Beschreibung	Standard
`bino.binPath`	Pfad zur `bino`-CLI-Binary, falls sie nicht im `PATH` liegt.	`"bino"`
`bino.enableCompletion`	Autovervollständigung für DataSet-Referenzen, Szenarien etc. aktivieren.	`true`
`bino.columnCacheTTL`	Cache-Dauer für Spalten-Introspection in Millisekunden.	`60000` (60 Sek.)
`bino.validateOnSave`	Workspace beim Speichern validieren und Lint-Ergebnisse im Problems-Panel anzeigen.	`true`

> **Warum eine Extension?** Ohne die Extension arbeitest du mit reinem YAML – ohne Validierung, ohne Autovervollständigung und ohne Navigation zwischen Manifesten. Die Extension verwandelt VS Code in eine auf BinoBI zugeschnittene Entwicklungsumgebung, die Tippfehler sofort erkennt, Referenzen auflöst und dir die verfügbaren Optionen vorschlägt. Das spart Zeit und reduziert Fehler erheblich.

---

02 Projekt initialisieren

02.01 Ein neues Projekt anlegen

Wechsle in das Verzeichnis, in dem du dein Berichtsprojekt ablegen möchtest, und führe aus:

```
bino init
```

Der Wizard fragt dich nach einem Zielverzeichnis, einem internen Projektnamen, einem Titel und der Sprache. Du kannst die Vorschläge übernehmen oder eigene Werte eingeben. Wenn du die Standardwerte ohne Rückfragen akzeptieren möchtest, verwende:

```
bino init -y
```

> **Warum ein Wizard?** `bino init` erzeugt nicht nur die Verzeichnisstruktur, sondern legt auch eine `bino.toml`-Konfigurationsdatei und Beispiel-Manifeste an. So hast du sofort ein funktionsfähiges Grundgerüst, das du erweitern kannst.

Wechsle anschließend in das neu erstellte Verzeichnis:

```
cd mein-bericht
```

02.02 Die Projektstruktur verstehen

`bino init` legt eine empfohlene Verzeichnisstruktur an. Diese Struktur ist keine zwingende Vorgabe – du kannst sie deinen Bedürfnissen anpassen. Wir empfehlen jedoch, bei der Standardstruktur zu bleiben, da alle Beispiele in diesem Handbuch darauf aufbauen.

```
mein-bericht/
├── bino.toml             # Projektkonfiguration
├── components/           # Wiederverwendbare Visualisierungskomponenten
├── datasets/             # DataSet-Manifeste (SQL-Abfragen)
├── datasources/          # DataSource-Manifeste (Datenverbindungen)
├── docs/                 # Markdown-Dateien verwendet von Dokument Artefakte
├── i18n/                 # Übersetzungstabellen (eine Datei je Sprache)
├── pages/                # Berichtsseiten (LayoutPage-Manifeste)
├── reports/              # Artefakt-Manifeste (PDF, Live, Screenshots)
├── resources/
│   ├── assets/           # Bilder, Logos, Fonts, Piktogramme
│   ├── data/             # Datendateien (CSV, Excel, Parquet)
│   └── sql/              # Externe SQL-Dateien
├── scripts/              # Skripte die während des Lifecycles aufgerufen werde
└── styles/               # Styling-Konfiguration (Farben, Schriften etc.)
```

Verzeichnis	Zweck
`components/`	Parametrisierte Visualisierungskomponenten (Tabelle, Chart), die auf mehreren Seiten wiederverwendet werden können.
`datasets/`	DataSet-Manifeste – SQL-Abfragen, die auf DataSources arbeiten. Die eigentlichen `.sql`-Dateien liegen in `resources/sql/`.
`datasources/`	DataSource-Manifeste – beschreiben, woher die Rohdaten kommen (CSV-Dateien, Datenbanken etc.).
`docs/`	Makrdown-Datein die von Dokument Artefakt verwendet werden.
`i18n/`	Übersetzungstabellen für die Internationalisierung. Jede Sprache bekommt eine eigene YAML-Datei.
`pages/`	Die einzelnen Berichtsseiten. Jede Seite ist ein `LayoutPage`-Manifest mit Kinder-Komponenten.
`reports/`	Artefakt-Manifeste (z. B. `ReportArtefact`), die festlegen, welche Seiten zu einem PDF zusammengefügt werden.
`resources/`	Alle externen Ressourcen: Datendateien, Bilder, SQL-Skripte, Fonts.
`scripts/`	Alle Skripte die während des Lifecycles aufgerufen werden können.
`styles/`	Styling-Dateien, um Farben, Schriftgrößen, Balkenbreiten usw. anzupassen.

02.03 Die bino.toml

Im Projektstamm liegt die Datei `bino.toml`. Sie markiert das Verzeichnis als Bino-Projekt und enthält unter anderem eine eindeutige Projekt-ID:

```
report-id = "550e8400-e29b-41d4-a716-446655440000"
```

In späteren Kapiteln wirst du hier auch Standard-Build-Parameter und Umgebungsvariablen konfigurieren können. Für den Moment reicht die Standardkonfiguration.

> **Warum `report-id`?** Die report-id muss eindeutig sein, darf aber ein beliebiger String sein. Die Id wird für die spätere eindeutige Identifizierung für Caching oder Scheduling verwendet. Dazu später mehr.

---

03 Eine DataSource anlegen

Grundlage jedes Berichts sind die **Daten**. Der erste Baustein ist die *DataSource*. Eine DataSource beschreibt, woher die Rohdaten stammen – zum Beispiel aus einer CSV-Datei, einer Excel-Tabelle oder einer Datenbank.

Für unser erstes Beispiel verwenden wir eine einfache CSV-Datei.

03.01 Die CSV-Datei bereitstellen

Der Wizard erstellt die Datei `resources/data/bb_example_data.csv` mit Beispieldaten, die wir zum Erstellen von Berichten verwenden können. Alternativ kannst du auch eine eigene Datenquelle (Datenbank oder Datei) verwenden.

> **Warum `resources/data/` ?** Die Konvention, Datendateien in `resources/data/` abzulegen, hält das Projekt übersichtlich – Daten sind klar von Manifest-Dateien getrennt. BinoBI erwartet kein bestimmtes Verzeichnis, aber wir empfehlen diese Standardstruktur.

03.02 Das DataSource-Manifest mit dem Wizard erstellen

Statt die YAML-Datei von Hand zu schreiben, nutze den interaktiven Wizard:

```
bino add datasource
```

Der Wizard fragt der Reihe nach (die Fragen erscheinen auf Englisch):

1. `Name for this DataSource` – sprechender Name in `snake_case`, z. B. `bb_example_data`.
2. `Description (optional)` – optional, aber empfohlen, z. B. *„Beispieldaten nach Region"*.
3. `What type of data source?` – Auswahlliste; wähle `CSV file`.
4. `File source` – entweder `Select existing file` (Fuzzy-Suche über die `.csv`-Dateien im Projekt, danach die Frage `Select file`) oder `Enter path manually` (danach `File path`). Wähle `resources/data/bb_example_data.csv`.
5. `Auto-detect CSV delimiter?` – antworte `no`, dann folgt `Delimiter character`; trage dort `;` ein. Mit `yes` erkennt BinoBI das Trennzeichen selbst und schreibt gar kein `delimiter` ins Manifest.
6. `CSV has header row?` – `yes`, die erste Zeile enthält die Spaltennamen.
7. `Rows to skip at beginning` – `0`.
8. `Add constraints to conditionally include this DataSource?` – `no` (Constraints folgen in Kapitel 105).
9. `Where to save the manifest?` – `Auto: datasources/<name>.yaml` (empfohlen), `Append to: …` oder `Custom path...`.

10. Der Wizard zeigt unter `=== Preview ===` das fertige Manifest und darunter `Will create: <Pfad>`. Erst wenn du `Proceed?` mit `yes` bestätigst, wird die Datei geschrieben.
11. `What next?` – `Done`, `Add a DataSet using this DataSource`, `Add another DataSource` oder `Run lint`. Für dieses Beispiel `Done`.

03.03 Das erzeugte Manifest

Nach Abschluss des Wizards findest du eine Datei wie `datasources/bb_example_data.yaml`:

```
apiVersion: bino.bi/v1alpha1
kind: DataSource
metadata:
  name: bb_example_data
  description: Beispieldaten nach Region
spec:
  type: csv
  path: ../resources/data/bb_example_data.csv
  delimiter: ";"
```

> **Achte auf den Pfad.** `spec.path` wird relativ zum **Manifest** aufgelöst, nicht zum Projektwurzelverzeichnis. Aus `datasources/` heraus führt der Weg zu den Daten deshalb über `../resources/data/…`.

> **Tipp:** Verwendet deine CSV-Datei ein Komma als Dezimaltrennzeichen (z. B. `1.234,56`), füge dem Manifest die Zeile `decimalSeparator: ","` hinzu. Ohne diese Angabe interpretiert BinoBI Punkte als Dezimaltrennzeichen.

---

04 Ein DataSet anlegen

Eine DataSource stellt die Rohdaten bereit. Ein **DataSet** formt diese Daten per SQL-Abfrage in die Struktur, die deine Visualisierung benötigt. BinoBI bettet DuckDB als Query-Engine ein – du brauchst also keine laufende Datenbank.

Für unser erstes Beispiel verwenden wir ein einfaches *Pass-Through*-DataSet, das alle Daten der Quelle unverändert weiterreicht. Das ist der einfachste Einstieg; SQL-Transformationen behandeln wir in späteren Kapiteln.

> **Hinweis:** Die verwendete CSV-Datei entspricht bereits dem von BinoBI verwendeten Daten Schema, daher ist ein *Pass-Through* moglich und kein ETL nötig.

04.01 Das DataSet mit dem Wizard erstellen

```
bino add dataset
```

Der Wizard fragt:

1. `Name for this DataSet` – z. B. `bb_example_dataset`.
2. `Description (optional)` – z. B. *„Alle Beispieldaten"*.
3. `How do you want to define your data?` – Auswahlliste: `SQL (write inline)`, `SQL (from file)`, `PRQL (write inline)`, `PRQL (from file)`, `Pass-through`. Wähle `Pass-through`.
4. `Select DataSource` – Fuzzy-Suche über die vorhandenen DataSources; wähle `bb_example_data`.
5. `Add constraints to conditionally include this DataSet?` – `no`.
6. `Where to save the manifest?` – `Auto: datasets/<name>.yaml` (empfohlen).
7. Preview prüfen, `Proceed?` mit `yes` bestätigen.
8. `What next?` – für dieses Beispiel `Done`.

** Das erzeugte Manifest**

```
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: bb_example_dataset
  description: Alle Umsatzdaten
spec:
  source: bb_example_data
```

Ein Pass-Through-DataSet wird als `source:` geschrieben, nicht als Query. Die Abhängigkeit ergibt sich daraus von selbst, `dependencies:` entfällt.

> **Warum ein DataSet, wenn wir die Daten nur durchreichen?** In BinoBI sind DataSources und DataSets bewusst getrennt. Die DataSource beschreibt *wo* die Daten liegen; das DataSet beschreibt *was* davon im Bericht gezeigt wird. Selbst wenn du anfangs alles durchreichst, kannst du später Filter, Aggregationen oder Joins ergänzen, ohne die DataSource zu ändern.

04.02 Bino Report VSCode Extension - DataSet Preview

Nachdem du das Dataset beschrieben hast, kannst du in der ersten Zeile der yaml ein kleines Menü der Bino Report Extension sehen. Durch das Klicken auf Preview öffnet sich ein neues DataSet preview Fenster mit den ersten 100 Datenreihen.

04.03 `query` – die SQL-Abfrage im Detail

Das Feld `query` enthält eine SQL-Abfrage, die von der eingebetteten **DuckDB**-Engine ausgeführt wird. Du kannst den vollen SQL-Funktionsumfang von DuckDB nutzen – das umfasst deutlich mehr als nur `SELECT *`. Du kannst mehr über die Funktionsweise der DataSets in Kapitel 103 lesen.

04.04 `dependencies` – die Abhängigkeiten

Das Feld `dependencies` ist eine Liste von Namen, die angibt, welche **DataSources** oder **anderen DataSets** in der `query` referenziert werden. Durch die Beschreibung der dependencies kann BinoBI die Ausführungsreihenfolge sicherstellen.

BinoBI verarbeitet DataSources und DataSets nicht in einer festen Reihenfolge, sondern baut aus den `dependencies` einen **Abhängigkeitsgraphen** (DAG – Directed Acyclic Graph). Dadurch wird garantiert, dass eine Quelle oder ein vorgelagertes DataSet vollständig geladen ist, *bevor* die darauf aufbauende Abfrage ausgeführt wird.

Beispiel: Wenn DataSet `B` von DataSet `A` abhängt und DataSet `A` von DataSource `csv_quelle`, dann ist die Ausführungsreihenfolge:

```
csv_quelle  →  DataSet A  →  DataSet B
```

04.05 Was passiert unter der Haube?

Wenn BinoBI den Bericht erstellt, durchläuft es eine Pipeline:

```
DataSource  →  DataSet  →  LayoutPage  →  ReportArtefact  →  PDF
 (Rohdaten)    (SQL)       (Seite)        (Zusammenbau)     (Ausgabe)
```

Die DataSource registriert die CSV-Datei als benannte Tabelle in der Query-Engine. Das DataSet führt die SQL-Abfrage darauf aus und stellt das Ergebnis als weitere benannte Tabelle bereit, die von Seitenkomponenten referenziert werden kann.

---

05 Eine Berichtsseite erstellen

Jetzt haben wir Daten (DataSource) und eine aufbereitete Sicht darauf (DataSet). Im nächsten Schritt erstellen wir eine **Berichtsseite** – ein `LayoutPage`-Manifest, das festlegt, wie die Daten visuell dargestellt werden.

Unsere Seite soll einen Seitenkopf mit Titel und eine konfigurierte Tabelle enthalten.

05.01. Das LayoutPage-Manifest anlegen

Erstelle die Datei `pages/page_beispiel_tabelle.yaml` mit folgendem Grundgerüst:

```
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: page_beispiel_tabelle
  description: Einfache Tabelle
spec:
  pageOrientation: portrait
  pageFormat: a4
  children:
    - kind: Table
      spec:
        dataset: bb_example_dataset
```

Wenn keine Attribute explizit gewählt wurden, wird das gesamte Dataset mit allen Szenarien verwendet.

	PY	PY2	PY3	PY4	PL	PL2	PL3	PL4	FC	FC2	FC3	FC4	
Amsterdam	12.457,0	18.873,0	15.887,0	12.360,0	12.446,0	18.881,0	15.875,0	12.363,0	12.559,0	18.908,0	15.832,0	12.410,0	12.4
Auckland	12.390,0	18.847,0	15.901,0	12.452,0	12.428,0	18.855,0	15.915,0	12.437,0	12.442,0	18.872,0	15.924,0	12.474,0	12.3
Austin	16.400,0	16.904,0	14.768,0	16.831,0	17.116,0	14.942,0	15.487,0	17.116,0	17.267,0	14.952,0	15.526,0	17.223,0	16.9
Beijing	14.758,0	15.199,0	14.867,0	15.221,0	15.487,0	16.303,0	15.577,0	15.487,0	15.486,0	16.382,0	15.514,0	15.472,0	15.2
Chicago	16.486,0	17.013,0	11.463,0	17.008,0	17.299,0	14.673,0	12.048,0	17.299,0	17.441,0	14.761,0	12.041,0	17.359,0	17.0
Dubai	12.416,0	18.846,0	15.932,0	12.297,0	12.373,0	18.850,0	15.889,0	12.324,0	12.429,0	18.905,0	15.916,0	12.381,0	12.4
Frankfurt	13.005,0	13.412,0	16.714,0	13.442,0	13.678,0	21.555,0	17.569,0	13.678,0	13.717,0	21.577,0	17.536,0	13.765,0	13.4
Hong Kong	16.410,0	16.889,0	13.079,0	16.908,0	17.205,0	17.206,0	13.674,0	17.205,0	17.253,0	17.265,0	13.755,0	17.262,0	16.8
London	16.079,0	16.553,0	19.270,0	16.486,0	16.754,0	11.325,0	20.106,0	16.754,0	16.916,0	11.337,0	20.243,0	16.797,0	16.5
Los Angeles	14.463,0	14.912,0	16.007,0	14.806,0	15.126,0	9.603,0	16.753,0	15.126,0	15.210,0	9.651,0	16.808,0	15.127,0	14.9
Madrid	12.454,0	18.918,0	15.952,0	12.350,0	12.477,0	18.899,0	15.937,0	12.368,0	12.516,0	19.095,0	15.955,0	12.362,0	12.5
Melbourne	15.700,0	16.168,0	12.555,0	16.094,0	16.392,0	18.839,0	13.133,0	16.392,0	16.446,0	18.895,0	13.226,0	16.474,0	16.1
Mexico City	12.390,0	18.875,0	15.824,0	12.352,0	12.379,0	18.864,0	15.809,0	12.374,0	12.412,0	18.912,0	15.739,0	12.387,0	12.4
Milan & Lombardy	10.508,0	10.811,0	18.570,0	10.798,0	11.054,0	14.671,0	19.470,0	11.054,0	10.964,0	14.726,0	19.491,0	11.012,0	10.8
Mumbai	12.411,0	18.810,0	15.932,0	12.329,0	12.418,0	18.794,0	15.944,0	12.363,0	12.361,0	18.794,0	15.950,0	12.286,0	12.4
Munich	12.422,0	18.909,0	15.865,0	12.446,0	12.421,0	18.909,0	15.864,0	12.427,0	12.436,0	18.897,0	15.885,0	12.488,0	12.4
New York City	20.073,0	20.706,0	18.924,0	20.671,0	21.010,0	13.768,0	19.743,0	21.010,0	21.126,0	13.774,0	19.910,0	21.067,0	20.7
Paris	17.398,0	17.941,0	20.087,0	17.809,0	18.207,0	14.496,0	21.008,0	18.207,0	18.389,0	14.585,0	21.123,0	18.237,0	17.9
San Francisco	12.454,0	18.836,0	15.948,0	12.391,0	12.441,0	18.839,0	15.948,0	12.401,0	12.550,0	18.812,0	15.937,0	12.406,0	12.5
Seoul	12.359,0	18.859,0	15.893,0	12.383,0	12.408,0	18.879,0	15.909,0	12.338,0	12.415,0	18.782,0	15.775,0	12.430,0	12.3
Shanghai	21.600,0	22.257,0	13.438,0	22.230,0	22.550,0	13.857,0	14.132,0	22.550,0	22.764,0	13.883,0	14.062,0	22.616,0	22.2
Sidney	16.512,0	16.986,0	11.160,0	16.975,0	17.295,0	14.218,0	11.685,0	17.295,0	17.263,0	14.191,0	11.748,0	17.295,0	16.9
Singapore	9.878,0	10.181,0	20.113,0	10.217,0	10.325,0	15.397,0	21.012,0	10.325,0	10.429,0	15.521,0	21.064,0	10.486,0	10.1
Tokyo	18.285,0	18.807,0	15.860,0	18.868,0	19.110,0	15.214,0	16.574,0	19.110,0	19.130,0	15.230,0	16.635,0	19.277,0	18.8
Toronto	14.092,0	14.532,0	13.002,0	14.512,0	14.764,0	17.116,0	13.584,0	14.764,0	14.765,0	17.245,0	13.631,0	14.808,0	14.5
Würzburg	13.054,0	13.439,0	16.770,0	13.425,0	13.678,0	21.555,0	17.569,0	13.678,0	13.664,0	21.574,0	17.685,0	13.708,0	13.4
Zurich	12.379,0	18.829,0	15.914,0	12.480,0	12.373,0	18.843,0	15.916,0	12.485,0	12.455,0	18.803,0	15.888,0	12.504,0	12.4

Table 1: einfache Tabelle

05.01.01 Was bedeuten die einzelnen Felder?

Feld	Bedeutung
`pageOrientation`	Hochformat (`portrait`) oder Querformat (`landscape`).
`pageFormat`	Seitenformat, z. B. `a4` , `letter` oder `xga` (Bildschirmformat).
`children`	Liste der Komponenten auf der Seite – hier eine Tabelle (`Table`).
`dataset`	Der Name des DataSets, aus dem die Tabelle ihre Daten bezieht.

> **Warum YAML statt einer grafischen Oberfläche?** BinoBI verfolgt den Ansatz *Reporting as Code*: Jede Berichtsdefinition ist eine Textdatei, die du versionieren, reviewen und in CI/CD-Pipelines automatisieren kannst. Änderungen lassen sich per `git diff` nachvollziehen – genau wie bei Anwendungscode.

05.02 Den Seitenkopf konfigurieren

Im nächsten Schritt erweitern wir die LayoutPage-Konfiguration um einen Seitenkopf mit Organisationseinheit, Berichtszeitraum und Kennzahlen. Die vollständige Attribut-Referenz findest du in der API-Dokumentation (https://cli.bino.bi/reference/layout-page/).

```
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: page_beispiel_tabelle
  description: Einfache Tabelle
spec:
  pageOrientation: portrait
  pageFormat: a4
  titleBusinessUnit: "BinoBI – Example Department"
  titleDateStart: 2026-09-30
  titleDateEnd: 2026-09-30
  titleMeasures:
    - name: "Revenue"
      unit: "kEUR"
  titleScenarios: [ac1, pp1]
  titleDateFormat: quarter
  titleDateLink: end
  titleOrder: category
  titleOrderDirection: asc
  children:
    - kind: Table
      spec:
        dataset: bb_example_dataset
```

05.02.01 Was bedeuten die Titel-Felder?

Feld	Bedeutung
`titleBusinessUnit`	Name der Organisationseinheit, der im Seitenkopf angezeigt wird (z. B. Abteilung, Bereich).
`titleDateStart`	Startdatum des Berichtszeitraums im ISO-8601-Format (`YYYY-MM-DD`).
`titleDateEnd`	Enddatum des Berichtszeitraums im ISO-8601-Format.
`titleMeasures`	Liste der Kennzahlen, die im Titel angezeigt werden. Jeder Eintrag hat einen `name` (z. B. *Revenue*) und eine `unit` (z. B. *kEUR*).

Feld	Bedeutung
`titleScenarios`	Szenarien, die im Titel dargestellt werden. Typische Werte: `ac1` (Actual), `fc1` (Forecast), `pp1` (Prior Year), `pl1` (Plan).
`titleDateFormat`	Anzeigeformat des Datums im Titel. Mögliche Werte: `year`, `quarter`, `month`, `week`, `day`, `time`, `auto`, `none`.
`titleDateLink`	Verknüpfungsstil des Datums. Mögliche Werte: `avg`, `interval`, `cum`, `start`, `end`, `ytd`, `ytg`, `mat`, `none`.
`titleOrder`	Sortierung der Titel-Elemente. Mögliche Werte: `category`, `categoryindex`, `rowgroup`, `rowgroupindex`, `ac1`, `fc1`, `pp1`, `pl1`.
`titleOrderDirection`	Sortierrichtung: `asc` (aufsteigend) oder `desc` (absteigend).

05.03 Die Tabelle konfigurieren

Im nächsten Schritt konfigurieren wir die Tabelle im Detail. Du kannst festlegen, welche Szenarien und Kennzahlen dargestellt werden, wie die Werte formatiert werden und ob Abweichungen als Inline-Balkendiagramme visualisiert werden sollen.

```
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: page_beispiel_tabelle
  description: Einfache Tabelle
spec:
  pageOrientation: portrait
  pageFormat: a4
  titleBusinessUnit: "BinoBI – Example Department"
  titleDateStart: 2026-09-30
  titleDateEnd: 2026-09-30
  titleMeasures:
    - name: "Revenue"
      unit: "kEUR"
  titleScenarios: [ac1, pp1]
  titleDateFormat: quarter
  titleDateLink: end
  titleOrder: category
  titleOrderDirection: asc
  children:
```

```
  - kind: Table
    spec:
      dataset: bb_example_dataset
      sumTitle: All Shop Locations
      level: category
      order: ac1
      orderDirection: desc
      measureScale: k
      measureType: currency
      measureUnit: EUR
      limit: 10
      unitScaling: 600
      scenarios: [ac1, pl1]
      variances: [dac1_pp1_pos, drac1_pp1_pos]
      grouped: false
      showGroupTitle: true
      showMeasureScale: true
      barColumns: [dac1_pp1_pos, drac1_pp1_pos]
      percentageScaling: 1
      dataFormat: decimal
      dataFormatDigitsDecimal: 0
      dataFormatDigitsPercent: 1
      type: sum
      categoryWidth: w20
```

in kEUR	PL	AC↓	ΔPY	ΔPY%
Shanghai	23	22	+0,7	*+3,1%*
New York City	21	21	+0,6	*+3,2%*
Tokyo	19	19	+0,5	*+2,8%*
Paris	18	18	+0,5	*+3,1%*
Chicago	17	17	+0,5	*+3,2%*
Sidney	17	17	+0,5	*+2,9%*
Austin	17	17	+0,5	*+3,1%*
Hong Kong	17	17	+0,5	*+2,9%*
London	17	17	+0,5	*+3,0%*
Melbourne	16	16	+0,5	*+2,9%*
REST	218	217	+3,1	*+1,4%*
All Shop Locations	**401**	**397**	**+8**	***+2,2%***

Table 2: erste konfigurierte Tabelle

05.04 Was bedeuten die Tabellen-Felder?

Feld	Bedeutung
`dataset`	Pflichtfeld – Name des DataSets, das die Tabellendaten liefert.
`sumTitle`	Beschriftung der Gesamtsummenzeile am Ende der Tabelle – **kein** Titel über der Tabelle. Wirkt nur bei `type: sum` und `type: opt`; ohne Angabe erscheint dort das IBCS-Summenzeichen ❖.
`level`	Hierarchie-Ebene, auf der die Tabelle anzeigt (z. B. `category`).
`order`	Spalte, nach der sortiert wird. Werte: `category`, `categoryindex`, `rowgroup`, `rowgroupindex`, `ac1` – `ac4`, `fc1` – `fc4`, `pp1` – `pp4`, `pl1` – `pl4`, `auto`.
`orderDirection`	Sortierrichtung: `asc` (aufsteigend) oder `desc` (absteigend).
`measureScale`	SI-Präfix für die Skalierung der Zahlenwerte. Werte: `_` (keine), `k` (Tausend), `M` (Millionen), `G` (Milliarden) u. a.
`measureType`	Art der Kennzahl: `volume` (Menge) oder `currency` (Währung).
`measureUnit`	Einheit, die im Tabellenkopf angezeigt wird (z. B. `EUR`, `USD`, `Stück`).
`limit`	Maximale Anzahl an Zeilen pro Gruppe. Überzählige Zeilen werden als „REST" zusammengefasst. `0` bedeutet unbegrenzt.
`unitScaling`	Dateneinheiten je 1 em Balkenlänge für absolute Abweichungsbalken (`d_`-Spalten). Map-Skala: ein größerer Wert ergibt kürzere Balken. Standard: `100`.
`scenarios`	Liste der Szenario-Spalten, die angezeigt werden (z. B. `[ac1, pl1]`). Mit `auto` werden Szenarien automatisch aus dem DataSet erkannt.
`variances`	Abweichungsdefinitionen im Format `d<SzenarioB>_<SzenarioA>_[pos|neg|neu]`, z. B. `dac1_pp1_pos` (absolute Abweichung, positiv = gut).
`grouped`	Zeilen nach der `rowGroup`-Dimension gruppieren (`true` / `false`).
`showGroupTitle`	Gruppenüberschriften anzeigen, wenn `grouped: true`.
`showMeasureScale`	Einheit und Skalierung im Tabellenkopf anzeigen (`true` / `false`).

Feld	Bedeutung
`barColumns`	Liste von Spaltennamen, die als IBCS-konforme Inline-Balken- oder Pin-Diagramme statt als Zahlenwerte dargestellt werden. Berücksichtigt werden nur Spalten, die auch in `scenarios` oder `variances` stehen; alles andere wird ohne Warnung ignoriert.
`percentageScaling`	Prozentpunkte je 1 em Pin-Länge für relative Abweichungs-Pins (`dr_` -Spalten). Ein größerer Wert ergibt kürzere Pins. Standard: `2` .
`dataFormat`	Zahlenformat: `decimal` (Dezimalzahl) oder `percent` (Prozent).
`dataFormatDigitsDecimal`	Anzahl der Nachkommastellen bei Dezimaldarstellung.
`dataFormatDigitsPercent`	Anzahl der Nachkommastellen bei Prozentdarstellung.
`type`	Tabellentyp: `list` (einfache Liste), `sum` (mit Summenzeile), `opt` (optimiert), `sumnototal` (Summe ohne Gesamtzeile), `optnototal` (optimiert ohne Gesamtzeile).
`categoryWidth`	Breite der Kategorie-Spalte. Werte: `w5` , `w10` , `w15` , `w20` , `w25` , `w50` , `inf` .

> **Tipp:** Die vollständige Attribut-Referenz für Tabellen findest du unter cli.bino.bi/reference/table/ (https://cli.bino.bi/reference/table/). Dort werden auch erweiterte Funktionen wie Drilldowns (`thereof` , `partOf` , `columnThereOf`) und Skalierung (`scale`) erklärt.

---

06 Das ReportArtefact anlegen

Eine Berichtsseite allein ergibt noch kein PDF. Dafür brauchst du ein **ReportArtefact** – es fasst eine oder mehrere Seiten zu einem fertigen Dokument zusammen und definiert Metadaten wie Dateiname, Titel und Sprache.

Erstelle die Datei `reports/mein_bericht.yaml` :

```
apiVersion: bino.bi/v1alpha1
kind: ReportArtefact
metadata:
  name: mein_bericht
spec:
```

```
  title: Mein erster Bericht
  description: Ein einfacher Umsatzbericht mit Tabelle
  filename: mein_bericht.pdf
  format: a4
  orientation: portrait
  language: de
  author: Max Mustermann
  layoutPages:
    - page: page_beispiel_tabelle
```

Feld	Bedeutung
`title`	Der Titel des PDF-Dokuments (erscheint in den PDF-Metadaten).
`filename`	Der Dateiname der erzeugten PDF-Datei.
`format`	Das Seitenformat des Artefakts.
`orientation`	Hoch- oder Querformat.
`layoutPages`	Liste der Seiten, die in das PDF aufgenommen werden. Jeder Eintrag referenziert eine `LayoutPage` über ihren Namen.

> **Warum ein separates Artefakt?** Die Trennung von Seite und Artefakt erlaubt es, dieselbe Seite in mehreren Berichten wiederzuverwenden – z. B. in einem Monatsbericht und einem Dashboard. Außerdem kannst du einer Seite beim Einbinden Parameter mitgeben, um sie mit unterschiedlichen Daten zu rendern (dazu mehr in späteren Kapiteln).

---

07 Vorschau im Browser

Bevor du das PDF erzeugst, kannst du deinen Bericht im Browser betrachten. BinoBI startet dafür einen lokalen Webserver mit Hot-Reload – Änderungen an Manifest- oder Datendateien werden sofort sichtbar.

```
bino preview
```

BinoBI scannt das Projektverzeichnis, verarbeitet alle Manifeste und öffnet deinen Standardbrowser unter `http://127.0.0.1:45678/`. Dort siehst du eine Übersicht aller Seiten und kannst sie einzeln durchblättern.

> **Tipp:** Mit `bino preview --log-sql` werden die ausgführten SQL-Abfragen im Terminal angezeigt – hilfreich, wenn du nachvollziehen möchtest, welche Daten eine Komponente erhält.

Beende die Vorschau mit `Strg+C` im Terminal.

> **Jetzt bist du dran!** Du hast bereits eine Vielzahl an Attributen kennengelernt – probiere sie aus! Starte mit `bino preview` die Vorschau und ändere parallel die YAML-Datei: Tausche die Sortierung, blende Szenarien ein oder aus, passe die Skalierung an oder experimentiere mit verschiedenen Tabellentypen. Dank Hot-Reload siehst du jede Änderung sofort im Browser. Keine Sorge, etwas kaputtzumachen – sollte ein ungültiger Zustand entstehen, warnt dich der Linter unter anderem durch ein oranges Ausrufezeichen im Preview-Fenster.

---

08 Fehler finden mit dem Linter

BinoBI bringt einen eingebauten Linter mit, der deine Manifest-Dateien gegen das Schema prüft und auf häufige Fehler hinweist – noch bevor du ein PDF erzeugst.

08.01 Manifest-Struktur prüfen

```
bino lint
```

Der Linter validiert die YAML-Struktur aller Manifeste: fehlende Pflichtfelder, ungültige Feldnamen, falsche Typen, nicht aufgelöste Referenzen und mehr. Die Ausgabe zeigt Warnungen und Fehler mit Dateiname und Zeilennummer.

08.02 Auch die Daten prüfen

Standardmäßig prüft `bino lint` nur die Struktur der Manifeste, ohne Abfragen tatsächlich auszuführen. Wenn du auch die Daten validieren möchtest – z. B. ob die SQL-Abfragen korrekt sind und die Ergebnis-Spalten zum erwarteten Schema passen –, verwende:

```
bino lint --execute-queries
```

Diese *Deep Validation* führt alle DataSet-Abfragen aus und prüft die Ergebnisse auf Typ-Fehler, ungültige Enum-Werte und fehlende Pflichtfelder.

> **Warum den Linter nutzen?** Der Linter erspart dir zeitaufwendige Build-Fehler. Er meldet Probleme sofort und präzise. Gewöhne dir an, `bino lint` regelmäßig auszuführen –

> idealerweise vor jedem Build. In CI-Pipelines kannst du mit `bino lint --execute-queries --fail-on-warnings` den Build automatisch fehlschlagen lassen, wenn Warnungen auftreten.

> **Gut zu wissen:** `bino build` führt den Linter standardmäßig automatisch aus. In der Vorschau (`bino preview`) ist er per Standard deaktiviert, kann aber mit `--lint` aktiviert werden.

---

09 Das PDF erzeugen

Wenn die Vorschau gut aussieht und der Linter keine Fehler meldet, ist es Zeit für das fertige PDF:

```
bino build
```

BinoBI durchläuft die gesamte Pipeline: Manifeste validieren, SQL-Abfragen ausführen, HTML rendern und in ein PDF konvertieren. Die Ausgabe landet standardmäßig im Verzeichnis `dist/` .

```
# Ausgabe prüfen
ls dist/
# → mein_bericht.pdf
```

Wenn du nur ein bestimmtes Artefakt bauen möchtest (z. B. wenn du mehrere Reports im Projekt hast):

```
bino build --artefact mein_bericht
```

> **Tipp:** Mit `bino build --log-sql` siehst du die SQL-Abfragen in der Build-Ausgabe. Mit `--out-dir` kannst du ein anderes Ausgabeverzeichnis angeben.

---

10 Zusammenfassung

In diesem Kapitel hast du den grundlegenden Arbeitsablauf mit Bino BI kennengelernt:

1. **Installation** – BinoBI CLI installieren und mit `bino setup` einrichten.
2. **Projekt anlegen** – Mit `bino init` ein neues Projekt mit Standardstruktur erzeugen.

3. **DataSource** – Eine CSV-Datei als Datenquelle registrieren (mit `bino add datasource`).
4. **DataSet** – Ein Pass-Through-DataSet anlegen, das die Rohdaten weiterreicht (mit `bino add dataset`).
5. **LayoutPage** – Eine Berichtsseite mit Titel und Tabelle definieren.
6. **ReportArtefact** – Die Seite in ein PDF-Artefakt einbinden.
7. **Vorschau** – Mit `bino preview` den Bericht im Browser betrachten.
8. **Linter** – Mit `bino lint` Fehler frühzeitig erkennen.
9. **Build** – Mit `bino build` das fertige PDF erzeugen.

Die gesamte Berichtsdefinition besteht aus Textdateien – YAML-Manifeste und SQL-Abfragen. Alles lässt sich versionieren, reviewen und automatisieren.

Im nächsten Kapitel vertiefen wir die Arbeit mit SQL-Abfragen und zeigen, wie du Daten filtern, aggregieren und für verschiedene Visualisierungen aufbereiten kannst.

104 - Die Vorschau mit bino preview

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute und Flags sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/cli/preview/ (https://cli.bino.bi/cli/preview/).

In den bisherigen Kapiteln hast du bereits `bino preview` verwendet, um Änderungen an deinem Bericht live im Browser zu sehen. Dabei haben wir den Befehl ohne weitere Optionen aufgerufen – einfach `bino preview` und fertig.

Dieses Kapitel erklärt ausführlich, was hinter `bino preview` steckt und wie du die Vorschau optimal für deinen Entwicklungsworkflow nutzt. Du lernst:

- was `bino preview` genau macht und wie der lokale Server funktioniert,
- welche Flags und Optionen zur Verfügung stehen,
- wie Hot-Reload funktioniert und welche Dateien überwacht werden,
- wie du Standardwerte und Umgebungsvariablen über die `bino.toml` konfigurierst,
- wie du Lint-Regeln in der Vorschau aktivierst,
- wie Datenvalidierung in der Vorschau funktioniert,
- wie du SQL-Logging für Debugging einsetzt,
- wie Hooks die Vorschau erweitern,
- wie Constraints mit `mode==preview` die Entwicklung erleichtern,
- wie Umgebungsvariablen in Manifesten aufgelöst werden,
- und worin sich `bino preview` von `bino build` und `bino serve` unterscheidet.

---

01 Was macht bino preview?

`bino preview` startet einen lokalen HTTP-Server, der deinen Bericht im Browser rendert. Der entscheidende Vorteil gegenüber `bino build`: Die Vorschau überwacht alle Manifest- und Datendateien in deinem Projekt und lädt die Ansicht automatisch neu, sobald du eine Datei änderst.

Der typische Workflow sieht so aus:

```
Terminal                          Browser
  |                                 |
```

```
  |  bino preview                  |
  |  --> Server startet            |
  |  --> Browser öffnet sich       |
  |                           Bericht wird angezeigt
  |                                |
  |  YAML-Datei speichern          |
  |  --> Änderung erkannt          |
  |  --> Neu-Render                |
  |                           Ansicht aktualisiert sich
  |                                |
  |  SQL-Datei speichern           |
  |  --> Änderung erkannt          |
  |  --> Queries neu ausführen     |
  |                           Daten aktualisiert
```

> **Wichtig:** `bino preview` ist ein Entwicklungswerkzeug. Es ist nicht für den produktiven Einsatz gedacht. Für produktive Szenarien verwende `bino build` (PDF-Erzeugung) oder `bino serve` (interaktive Web-Anwendung).

---

02 Grundlegende Verwendung

Im folgenden Abschnitt besprechen wir die Terminal-Befehle. Alternativ kannst du die Funktionen auch direkt über die `Bino Report` -Extension in Visual Studio Code verwenden:

https://marketplace.visualstudio.com/items?itemName=BinoBI.vscode-bino
(https://marketplace.visualstudio.com/items?itemName=BinoBI.vscode-bino)

02.01 Vorschau starten

Im einfachsten Fall genügt ein einziger Befehl im Projektverzeichnis:

```
bino preview
```

BinoBI startet einen lokalen HTTP-Server auf Port `45678` und öffnet automatisch den Browser unter `http://127.0.0.1:45678/` .

> **Tipp:** Sollte sich der Browser nicht automatisch öffnen, navigiere manuell zu `http://127.0.0.1:45678/` .

02.02 Vorschau beenden

Drücke `Ctrl+C` im Terminal, um den Preview-Server zu stoppen.

02.03 Anderes Projektverzeichnis

Wenn du die Vorschau für ein Projekt in einem anderen Verzeichnis starten möchtest, verwende das `--work-dir`-Flag:

```
bino preview --work-dir ./mein-bericht
```

---

03 Flags und Optionen

`bino preview` unterstützt mehrere Flags, die das Verhalten des Preview-Servers steuern. Die vollständige Syntax lautet:

```
bino preview [flags]
```

03.01 Übersicht aller Flags

Flag	Beschreibung	Standard
`--work-dir`	Projektverzeichnis (Report-Bundle).	`.`
`--port`	HTTP-Port des Preview-Servers.	`45678`
`--log-sql`	SQL-Abfragen im Terminal protokollieren.	`false`
`--lint`	Lint-Regeln bei jedem Refresh ausführen.	`false`
`--data-validation`	Datenvalidierungsmodus: `warn`, `fail` oder `off`.	`warn`
`--verbose`, `-v`	Ausführliche Logausgabe inkl. Run-ID aktivieren.	`false`
`--no-color`	Farbige Terminalausgabe deaktivieren (auch via `NO_COLOR`-Env).	`false`

Die einzelnen Flags werden in den folgenden Abschnitten dieses Kapitels genauer erklärt.

03.02 Port ändern

Wenn Port `45678` bereits belegt ist oder du einen anderen Port bevorzugst:

```
bino preview --port 9000
```

Der Bericht ist dann unter `http://127.0.0.1:9000/` erreichbar.

03.03 Mehrere Flags kombinieren

Flags lassen sich beliebig kombinieren:

```
bino preview --port 9000 --log-sql --lint --data-validation=fail
```

---

04 Hot-Reload: Automatisches Neuladen

Das Kernstück von `bino preview` ist das Hot-Reload. Die Vorschau überwacht dein Projektverzeichnis und reagiert automatisch auf Änderungen:

04.01 Welche Dateien werden überwacht?

- **YAML-Manifeste** – Änderungen an `LayoutPage`, `Table`, `ChartStructure` und allen anderen Manifest-Typen lösen ein Neuladen aus.
- **Externe Dateien** – Externe SQL-, PRQL-, CSV-, Excel- und Parquet-Dateien, die per `$file`-Syntax referenziert werden, sind vollständig in das Hot-Reload integriert. Eine Änderung an der Datei führt automatisch zum Neu-Ausführen der betroffenen DataSets.
- **Asset-Dateien** – Bilder und andere Assets, die per `Asset` eingebunden sind.

04.02 Workflow: Bearbeiten, Speichern, Sehen

Der Hot-Reload-Zyklus läuft vollständig automatisch:

1. Du änderst eine Datei in deinem Editor (z. B. VS Code).
2. Du speicherst die Datei.
3. `bino preview` erkennt die Änderung.
4. Die betroffenen DataSets und Komponenten werden neu berechnet.
5. Der Browser aktualisiert die Ansicht automatisch.

> **Tipp:** Kombiniere `bino preview` mit einem Editor, der automatisch speichert (z. B. VS Code mit `files.autoSave`). So siehst du Änderungen nahezu in Echtzeit.

---

05 SQL-Logging mit –log-sql

Das Flag `--log-sql` protokolliert alle SQL-Abfragen, die BinoBI intern an die DuckDB-Engine sendet, im Terminal. Das ist besonders nützlich für:

- **Debugging** – wenn eine Abfrage unerwartete Ergebnisse liefert,
- **Verständnis** – um zu sehen, welche Abfragen BinoBI aus deinen Manifesten generiert,
- **Performance** – um zu erkennen, welche Abfragen ausgeführt werden und wie oft.

05.01 SQL-Logging aktivieren

```
bino preview --log-sql
```

Im Terminal erscheinen dann bei jedem Refresh die ausgeführten SQL-Abfragen:

```
[SQL] SELECT region AS category, SUM(amount) AS ac1 FROM umsatz_csv GROUP BY reg
[SQL] SELECT product AS category, revenue AS ac1, budget AS pl1 FROM produkt_ds
```

> **Hinweis:** `--log-sql` gibt die Abfragen nur im Terminal aus, nicht im Browser. Es hat keinen Einfluss auf das Rendering.

---

06 Lint-Regeln in der Vorschau

Standardmäßig führt `bino preview` keine Lint-Regeln aus – im Gegensatz zu `bino build`, das Lint automatisch anwendet. Du kannst Lint in der Vorschau explizit aktivieren:

```
bino preview --lint
```

06.01 Was bewirkt –lint?

Wenn `--lint` aktiv ist, validiert BinoBI bei jedem Hot-Reload-Zyklus deine Manifeste gegen das Schema und führt inhaltliche Lint-Regeln aus. Warnungen und Fehler erscheinen direkt im Terminal.

Das ist besonders nützlich, um Fehler früh zu erkennen – noch bevor du `bino build` ausführst.

06.02 Vergleich: Lint in preview vs. build

Aspekt	`bino preview`	`bino build`
Lint standardmäßig	Nein (Opt-in mit `--lint`)	Ja (Opt-out mit `--no-lint`)
Lint deaktivieren	Flag weglassen	`--no-lint`
Wirkung bei Fehlern	Warnung im Terminal	Build schlägt fehl

> **Tipp:** Eine vollständige Übersicht aller Lint-Regeln findest du unter https://cli.bino.bi/appendix/lint-rules/ (https://cli.bino.bi/appendix/lint-rules/).

Hinweis: Wir empfehlen, die Vorschau stets mit `--lint` auszuführen, um mögliche Probleme möglichst frühzeitig zu erkennen.

---

07 Datenvalidierung

`bino preview` validiert Abfrageergebnisse gegen das Standard-Datenschema (siehe Kapitel 103, Abschnitt 08). Die Datenvalidierung prüft die Korrektheit der SQL sowie ob Daten in der View vorhanden sind.

07.01 Validierungsmodi

Das Flag `--data-validation` steuert, wie BinoBI mit Validierungsfehlern umgeht:

Modus	Verhalten	Empfehlung
`warn`	Warnungen ausgeben und die Vorschau fortsetzen (Standard).	Für die tägliche Arbeit
`fail`	Validierungsfehler als fatal behandeln und den Refresh abbrechen.	Für strenges Entwickeln
`off`	Validierung komplett deaktivieren.	Bei Performance-Problemen

```
# Warnungen (Standard)
bino preview --data-validation=warn

# Strenger Modus
bino preview --data-validation=fail
```

```
# Validierung deaktivieren
bino preview --data-validation=off
```

07.02 Was wird geprüft?

Prüfung	Beschreibung
Typvalidierung	String-Felder müssen Strings sein, Zahlenfelder müssen Zahlen sein.
Enum-Validierung	`operation` muss `"+"` oder `"-"` sein.
Datumsformat	`date` muss im ISO-8601-Format vorliegen (`YYYY-MM-DD`).
Abhängige Pflichtfelder	`rowGroup` erfordert `rowGroupIndex` , `category` erfordert `categoryIndex` usw.

07.03 Stichprobengröße anpassen

Aus Effizienzgründen validiert BinoBI standardmäßig nur die ersten 10 Zeilen pro DataSet. Die Größe lässt sich über eine Umgebungsvariable anpassen:

```
BNR_DATA_VALIDATION_SAMPLE_SIZE=100 bino preview
```

Umgebungsvariablen werden über die `bino.toml` verwaltet – Details dazu findest du in der Referenz (https://cli.bino.bi/cli/build/#environment-variables-via-binotoml). Die konkreten Konfigurationsmöglichkeiten der `bino.toml` werden im nächsten Kapitel ausführlich behandelt.

---

08 Konfiguration über bino.toml

Statt Flags jedes Mal auf der Kommandozeile anzugeben, kannst du Standardwerte dauerhaft in der `bino.toml` hinterlegen. Das spart Tipparbeit und sorgt dafür, dass alle Teammitglieder dieselben Einstellungen verwenden.

08.01 Standard-Flags setzen

Im Abschnitt `[preview.args]` der `bino.toml` definierst du Standardwerte für die Preview-Flags:

```
[preview.args]
port = 9000
log-sql = true
```

```
lint = true
data-validation = "warn"
```

Mit dieser Konfiguration startet `bino preview` automatisch auf Port 9000, mit SQL-Logging und Lint aktiviert – ohne weitere Flags auf der Kommandozeile.

08.02 Kommandozeile überschreibt bino.toml

Explizite Flags auf der Kommandozeile haben immer Vorrang vor den TOML-Standardwerten. BinoBI protokolliert dabei die Überschreibung:

```
$ bino preview --port 8080
Overriding port from bino.toml (9000 -> 8080)
```

08.03 Umgebungsvariablen setzen

Im Abschnitt `[preview.env]` kannst du Umgebungsvariablen definieren, die nur während der Vorschau gelten. Das ist besonders nützlich für Query-Limits oder Datenbankverbindungen in der Entwicklung:

```
[preview.env]
BNR_MAX_QUERY_ROWS = "10000"
BNR_CDN_MAX_BYTES = "10485760"
DATABASE_URL = "postgres://localhost/dev"
```

Auch hier gilt: Tatsächliche Umgebungsvariablen überschreiben die TOML-Werte:

```
$ BNR_MAX_QUERY_ROWS=5000 bino preview
Environment variable BNR_MAX_QUERY_ROWS overrides bino.toml ("10000" -> "5000")
```

> **Gut zu wissen:** Die vollständige Referenz zur `bino.toml` findest du unter https://cli.bino.bi/concepts/project-configuration/ (https://cli.bino.bi/concepts/project-configuration/) und Referenz (https://cli.bino.bi/cli/build/#environment-variables-via-binotoml).

---

09 Hooks: Automatisierung in der Vorschau

Hooks sind Shell-Befehle, die zu bestimmten Zeitpunkten im Preview-Lebenszyklus automatisch ausgeführt werden. Du definierst sie in der `bino.toml`. Jeder Checkpoint kann ein Array von

Aufrufen ausführen.

09.01 Verfügbare Checkpoints für die Vorschau

Checkpoint	Wann wird er ausgeführt?
`pre-preview`	Einmalig nach dem Start des Servers, vor dem ersten Laden.
`pre-refresh`	Vor jedem Hot-Reload-Zyklus.
`pre-datasource`	Vor dem Rendering der HTML-Ausgabe, jeweils pro Artefakt.

09.02 Beispiel: Testdaten vor der Vorschau generieren

```
[preview.hooks]
pre-preview = ["bino update", "python scripts/generate_fixtures.py"]
```

Dieses Skript wird einmalig beim Start der Vorschau ausgeführt – ideal für das Erzeugen von Testdaten oder das Herunterladen aktueller Quelldaten.

09.03 Beispiel: Daten vor jedem Refresh aktualisieren

```
[preview.hooks]
pre-refresh = ["python scripts/sync_data.py"]
```

Dieses Skript läuft vor jedem Hot-Reload-Zyklus. Damit kannst du z. B. Daten aus einer API abrufen, bevor die Vorschau neu gerendert wird.

09.04 Fehlerbehandlung bei Hooks

Hooks in der Vorschau sind fehlertolerant:

Hook-Typ	Verhalten bei Fehler
`pre-preview`	Die Vorschau startet nicht (fataler Fehler).
`pre-refresh`	Der Refresh wird übersprungen, der Fehler wird geloggt, der Server läuft weiter.
`pre-datasource`	Der Refresh wird übersprungen, der Fehler wird geloggt, der Server läuft weiter.

> **Hinweis:** Per-Cycle-Hooks wie `pre-refresh` sind bewusst fehlertolerant. Ein fehlschlagender Hook crasht den Preview-Server nicht – du kannst das Problem beheben und der nächste Refresh funktioniert wieder.

09.05 Umgebungsvariablen in Hooks

Jeder Hook erhält automatisch Umgebungsvariablen, die den aktuellen Kontext beschreiben:

Variable	Beschreibung	Beispiel
`BINO_MODE`	Aktueller Befehl.	`preview`
`BINO_WORKDIR`	Absoluter Pfad zum Projektverzeichnis.	`/home/user/reports`
`BINO_REPORT_ID`	Report-ID aus `bino.toml`.	`550e8400-...`
`BINO_HOOK`	Name des aktuellen Checkpoints.	`pre-refresh`
`BINO_LISTEN_ADDR`	Adresse, unter der der Server lauscht.	`127.0.0.1:45678`
`BINO_REFRESH_REASON`	Ursache für den Refresh (nur bei `pre-refresh`).	Dateipfad

09.06 Exit Code 78: Hook überspringen

Ein Hook kann den Exit Code `78` zurückgeben, um sich selbst zu überspringen. BinoBI behandelt das als Erfolg und fährt anschließend normal fort:

```
#!/bin/bash
# Daten nur aktualisieren, wenn sie aelter als 1 Stunde sind
if [ -f data.csv ] && [ $(($(date +%s) - $(stat -f %m data.csv))) -lt 3600 ]; th
  exit 78
fi
curl -o data.csv https://api.example.com/data
```

> **Hinweis:** Die vollständige Hook-Dokumentation mit allen Checkpoints und Umgebungsvariablen findest du unter https://cli.bino.bi/concepts/hooks/ (https://cli.bino.bi/concepts/hooks/).

---

10 Constraints mit mode==preview

Mit `metadata.constraints` kannst du Manifeste an bestimmte Modi binden – z. B. nur für die Vorschau oder nur für den Build. Für die Vorschau ist das Constraint `mode==preview` besonders nützlich, um Sampling, Debug-Ansichten oder Testdaten auf die Entwicklungsumgebung zu beschränken.

> **Hinweis:** Eine ausführliche Erklärung des Constraint-Systems mit allen Operatoren, Feldern und Beispielen findest du in Kapitel 105.

---

11 Umgebungsvariablen in Manifesten

BinoBI unterstützt Umgebungsvariablen in allen String-Feldern deiner Manifeste. Das Verhalten unterscheidet sich dabei zwischen Vorschau und Build:

11.01 Syntax

Syntax	Beschreibung
`${VAR}`	Wird durch den Wert der Variable `VAR` ersetzt.
`${VAR:default}`	Wird durch den Wert von `VAR` ersetzt oder durch `default`, falls `VAR` nicht gesetzt ist.
`\${VAR}`	Escape-Sequenz: erzeugt das Literal `${VAR}`.

11.02 Verhalten in preview vs. build

Szenario	`bino preview`	`bino build`
Variable nicht aufgelöst	Warnung, ein leerer String wird eingesetzt.	Build schlägt mit einem Fehler fehl.
Variable mit Standardwert	Standardwert wird verwendet.	Standardwert wird verwendet.

Beispiel in einer DataSource:

```
spec:
  type: csv
  path: "${DATA_DIR:./data}/sales.csv"
```

In der Vorschau wird `./data/sales.csv` verwendet, wenn `DATA_DIR` nicht gesetzt ist. Im Build würde dasselbe passieren – solange die Fallback-Syntax `${VAR:default}` verwendet wird.

> **Tipp:** Verwende immer Standardwerte (`${VAR:default}`) für Variablen, die in der Vorschau benötigt werden. So funktioniert `bino preview` auch ohne gesetzte Umgebungsvariablen.

---

12 Unterschiede: preview vs. build vs. serve

BinoBI bietet drei Befehle, die deinen Bericht verarbeiten. Jeder hat einen anderen Zweck:

12.01 Vergleich auf einen Blick

Feature	`bino preview`	`bino build`	`bino serve`
Zweck	Entwicklung	PDF-Erzeugung	Produktiver Server
Dateiüberwachung	Ja	Nein	Nein
Hot-Reload	Ja	Nein	Nein
PDF-Ausgabe	Nein	Ja	Nein
Query-Parameter-Substitution	Nein	Nein	Ja
Navigation (Multi-Route)	Nein	Nein	Ja
Lint standardmäßig	Nein	Ja	Nein
Produktionstauglich	Nein	Ja	Ja
Standard-Port	`45678`	–	`8080`
Unaufgelöste Env-Variablen	Warnung	Fehler	–

12.02 Wann welchen Befehl verwenden?

```
Entwicklung                    Deployment
  |                              |
  |  bino preview                |  bino build --> PDF
  |  (iterieren, testen)         |  (CI/CD, manueller Export)
  |                              |
```

```
  |                              |  bino serve --> Web-App
  |                              |  (Dashboard, interaktive Berichte)
```

- Verwende **`bino preview`** während der aktiven Entwicklung, wenn du Layouts, Daten und Komponenten iterativ anpasst.
- Verwende **`bino build`** für die finale PDF-Erzeugung – in der CI/CD-Pipeline oder auf der Kommandozeile.
- Verwende **`bino serve`** für interaktive Web-Anwendungen mit Query-Parametern und Navigation (erfordert ein `LiveReportArtefact`).

> **Hinweis:** Ausführliche Informationen zu `bino build` findest du unter https://cli.bino.bi/cli/build/ (https://cli.bino.bi/cli/build/), zu `bino serve` unter https://cli.bino.bi/cli/serve/ (https://cli.bino.bi/cli/serve/).

---

13 Praxistipps für die Vorschau

13.01 Schnellere Iteration mit Sampling

Bei großen Datenmengen kann die Vorschau träge werden. Verwende `sample` in deinen DataSources, um nur einen Teil der Daten zu laden (siehe Abschnitt 10.02 und Kapitel 103, Abschnitt 02.10):

```
spec:
  type: parquet
  path: ./warehouse/*.parquet
  sample: 1000
```

13.02 SQL debuggen mit –log-sql

Wenn eine Tabelle oder ein Chart nicht die erwarteten Daten zeigt, starte die Vorschau mit `--log-sql` und prüfe die ausgeführten Abfragen:

```
bino preview --log-sql
```

13.03 Lint aktivieren für frühes Feedback

Aktiviere `--lint` in der Vorschau, um Schema-Fehler sofort zu sehen – nicht erst beim nächsten `bino build`:

```
bino preview --lint
```

Oder dauerhaft in der `bino.toml`:

```
[preview.args]
lint = true
```

13.04 Empfohlene `bino.toml` für die Entwicklung

Eine bewusst konfigurierte `bino.toml` kann deinen Entwicklungsworkflow deutlich verbessern:

```
[preview.args]
port = 9000
log-sql = true
lint = true
data-validation = "warn"

[preview.env]
BNR_MAX_QUERY_ROWS = "10000"

[preview.hooks]
pre-preview = ["bino update","python scripts/generate_test_data.py"]
```

13.05 VS Code und bino preview

Die BinoBI VS Code-Extension erkennt die `bino.toml` im Workspace und aktiviert sich automatisch. In Kombination mit `bino preview` ergibt sich ein nahtloser Workflow:

1. Öffne dein Projekt in VS Code.
2. Starte `bino preview` im integrierten Terminal.
3. Bearbeite Manifeste – die Extension bietet Schema-Validierung.
4. Speichere – die Vorschau aktualisiert sich automatisch.

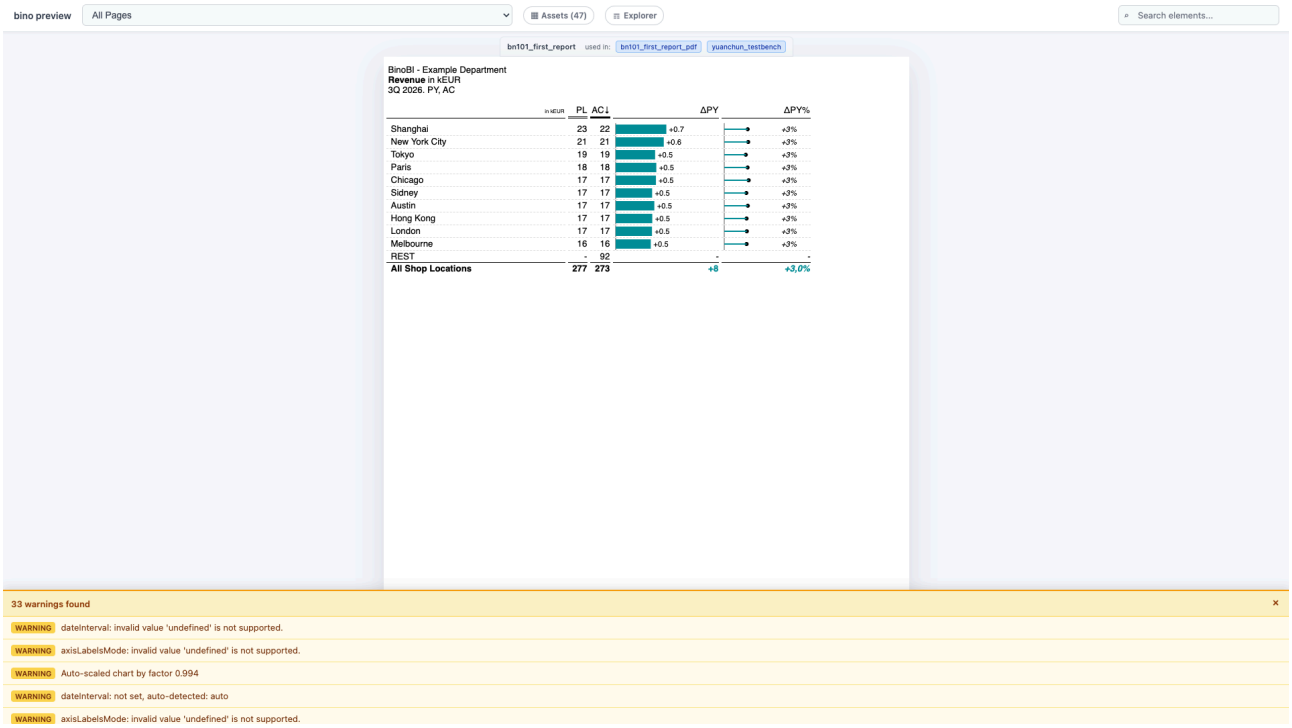
---

14 Die Benutzeroberfläche der Vorschau

Die Vorschau von `bino preview` ist nicht nur ein passives Rendering deines Berichts – sie bietet eine vollständige Entwicklungsumgebung im Browser. In diesem Abschnitt lernst du die einzelnen Bereiche der Preview-GUI und ihre Funktionen kennen.

14.01 Übersicht

Wenn du `bino preview` startest und `http://127.0.0.1:45678/` im Browser öffnest, siehst du die Preview-Oberfläche:



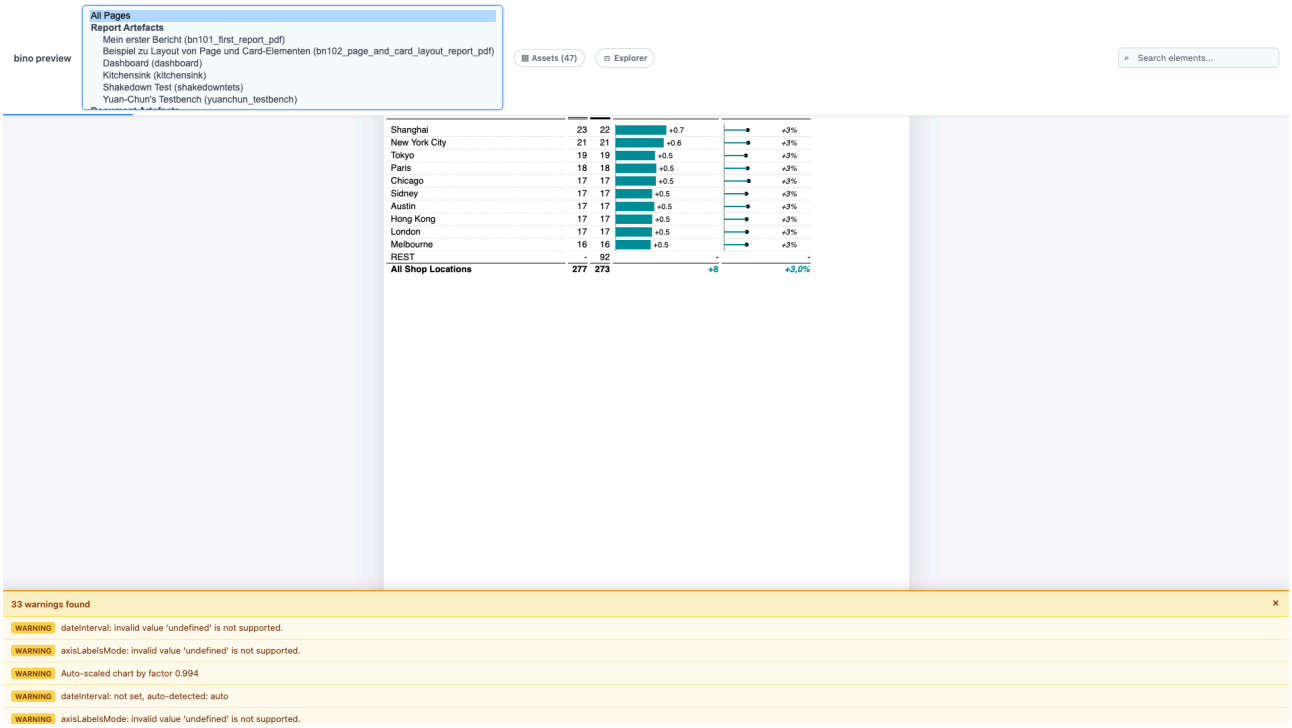
Die Oberfläche gliedert sich in folgende Bereiche:

Bereich	Position	Funktion
Toolbar	Oberer Rand	Artefakt-Dropdown, Assets-Button, Explorer-Button, Search
Bericht	Zentral	Rendering der aktuellen Seite(n)
Seiteninfo	Über der Seite	Nur in der Ansicht **All Pages**: zeigt den Namen der Seite und die Artefakte, in denen sie verwendet wird
Warning-Panel	Unterer Rand	Lint- und Validierungswarnungen (falls vorhanden)

Die Toolbar am oberen Rand ist das zentrale Steuerelement. Von links nach rechts findest du dort: den Titel `bino preview`, das Artefakt-Dropdown, den Assets-Button, den Explorer-Button und rechts die Suchleiste.

14.02 Artefakt-Dropdown

Das Dropdown-Menü oben links ist der Ausgangspunkt für die Navigation. Es listet alle Artefakte auf, die in deinem Projekt definiert sind:



Die Artefakte werden nach Typ gruppiert:

- **Report Artefacts** (`kind: ReportArtefact`) – klassische mehrseitige Berichte, die als PDF exportiert werden können.
- **Document Artefacts** (`kind: DocumentArtefact`) – Dokumenten-Artefakte, z. B. für längere Handbücher oder Berichte mit Inhaltsverzeichnis.

Zusätzlich steht die Option **All Pages** zur Verfügung. Sie zeigt alle `LayoutPage` -Manifeste des Projekts an – unabhängig davon, ob sie einem Artefakt zugeordnet sind. Das ist nützlich, wenn du an einer einzelnen Seite arbeitest, ohne den Kontext eines bestimmten Artefakts zu benötigen.

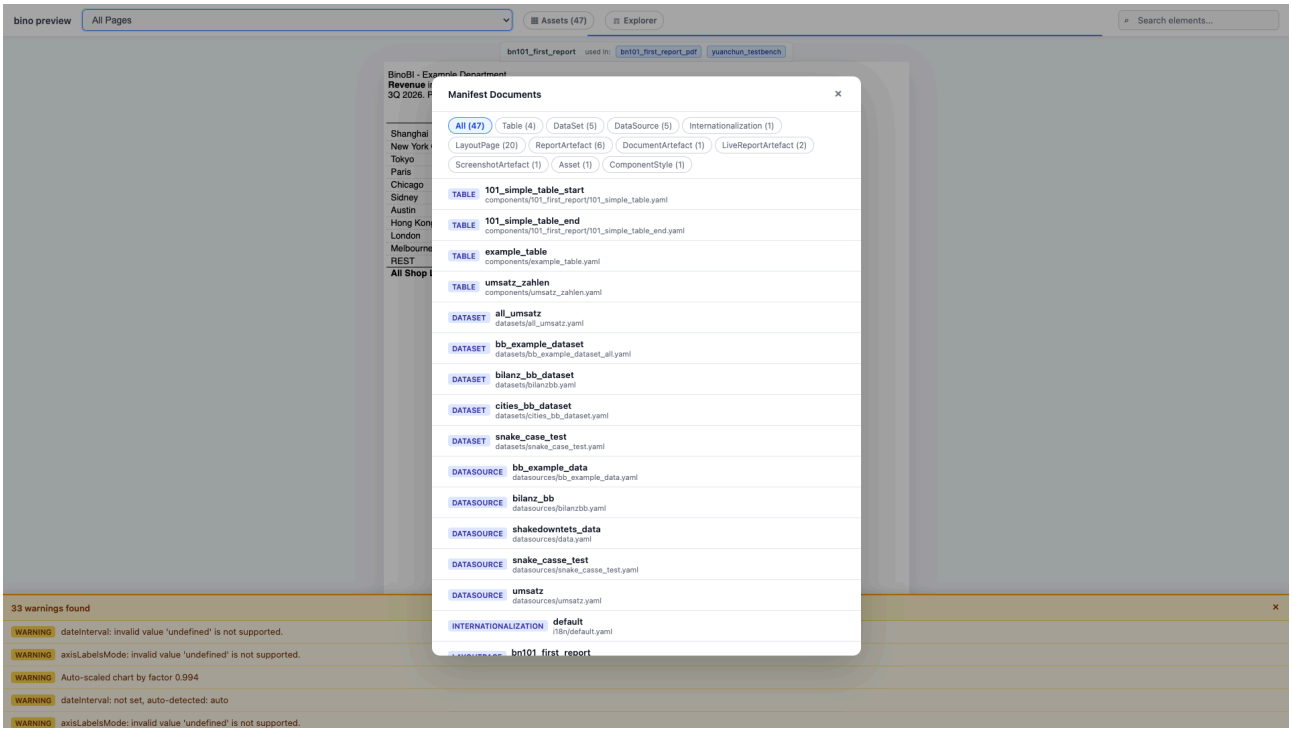
> **Tipp:** Wähle während der Entwicklung gezielt das Artefakt aus, an dem du gerade arbeitest. So siehst du nur die relevanten Seiten und die Vorschau lädt schneller.

Oberhalb der gerenderten Seite kann die Vorschau den **Seiteninfo**-Bereich anzeigen. Dort stehen der Name der Seite (`metadata.name`) und die Namen aller Artefakte, in denen diese Seite verwendet wird.

> **Einschränkung:** Der Bereich erscheint zurzeit nur in der Ansicht **All Pages**. Wählst du im Dropdown ein einzelnes Artefakt aus, wird er nicht angezeigt.

14.03 Assets Modal

Der Button **Assets** in der Toolbar öffnet das Manifest-Documents-Modal. Die Zahl in Klammern (z. B. `Assets (47)`) zeigt die Gesamtanzahl aller Manifest-Dokumente im Projekt:



Das Modal zeigt alle Manifest-Dokumente deines Projekts, gruppiert nach `kind`. Über die Filter-Tabs am oberen Rand kannst du gezielt nach Typ filtern:

Filter	Zeigt
All	Alle Manifest-Dokumente
Table	Tabellen-Komponenten (`kind: Table`)
DataSet	DataSet-Definitionen (`kind: DataSet`)
DataSource	Datenquellen (`kind: DataSource`)
Internationalization	Übersetzungsdateien (`kind: Internationalization`)
LayoutPage	Seitenlayouts (`kind: LayoutPage`)
ReportArtefact	Report-Artefakte (`kind: ReportArtefact`)
DocumentArtefact	Dokument-Artefakte (`kind: DocumentArtefact`)
LiveReportArtefact	Live-Report-Artefakte (`kind: LiveReportArtefact`)
ScreenshotArtefact	Screenshot-Artefakte (`kind: ScreenshotArtefact`)
Asset	Assets wie Bilder, Fonts, Pictogramme (`kind: Asset`)
ComponentStyle	Report-Styles (`kind: ComponentStyle`)

Jeder Eintrag zeigt den `metadata.name` und den Dateipfad des zugehörigen YAML-Manifests. So behältst du auch in größeren Projekten den Überblick über alle Bausteine deines Berichts.

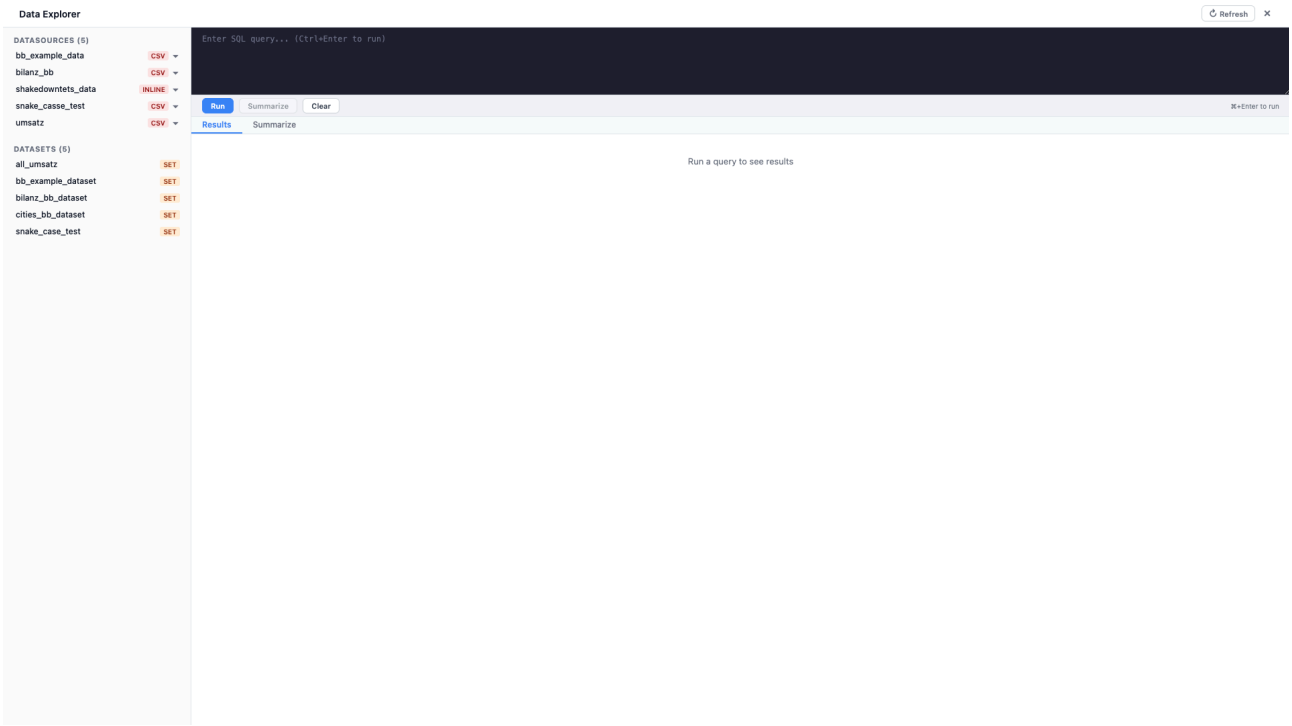
14.04 Dependency-Graph

Der **Graph**-Button in der Toolabr öffnet den Dependency-Graph, in dieser Anicht erhält man eine grafische Darstellung der in dem gewählten Artefakt vorhandenen Resourcen-Abghängigkeiten. Diese Ansicht hilft zu die Strukut des Artefaktes besser zu verstehen und ist eine ergänzung des Manifest Documents modasls.

> **Hinweis** Der Button ist nur sichtbar, wenn explizit ein Artefakt ausgewählt ist. Bei **All Pages** ist der Button ausgeblendet.

14.04 Data Explorer

Der **Explorer**-Button in der Toolbar öffnet den Data Explore mit dem du SQL-Abfragen direkt auf deinen Datenquellen und DataSets ausführen kannst:



Der Data Explorer besteht aus drei Bereichen:

Linke Seitenleiste – Datenquellen und DataSets:

Die Seitenleiste listet alle verfügbaren Daten auf, getrennt in zwei Gruppen:

- **DATASOURCES** – Alle `DataSource`-Manifeste des Projekts. Neben dem Namen wird der Typ angezeigt (z. B. `CSV` für CSV-Dateien, `INLINE` für Inline-Daten).
- **DATASETS** – Alle `DataSet`-Manifeste. Diese werden mit dem Badge `SET` gekennzeichnet.

Jeder Eintrag verfügt über ein Dropdown-Menü (Pfeil-Symbol), über das du weitere Aktionen ausführen kannst, wie z. B. den Spaltennamen in die Abfrage einzufügen.

Oberer Bereich – SQL-Editor:

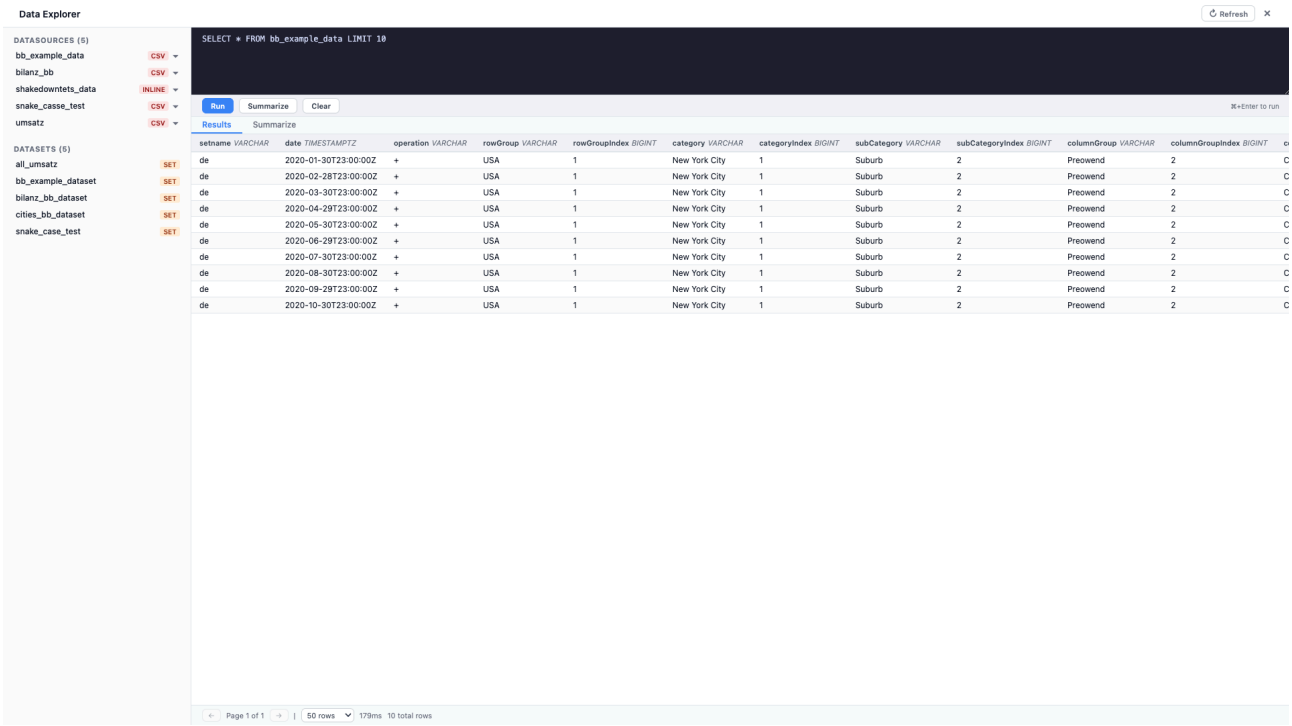
Im oberen Bereich befindet sich ein Textfeld, in das du beliebige SQL-Abfragen eingeben kannst. Die Abfragen werden direkt gegen die eingebettete DuckDB-Engine ausgeführt. Du kannst dabei alle SQL-Funktionen von DuckDB nutzen.

Die Steuerleiste unterhalb des Editors bietet folgende Buttons:

Button	Funktion
Run	Führt die SQL-Abfrage aus (Shortcut: `Ctrl+Enter` / `Cmd+Enter`)
Summarize	Erzeugt eine statistische Zusammenfassung der Ergebnisdaten
Clear	Leert den SQL-Editor

Unterer Bereich – Ergebnistabelle:

Nach dem Ausführen einer Abfrage werden die Ergebnisse in einer paginierten Tabelle angezeigt:

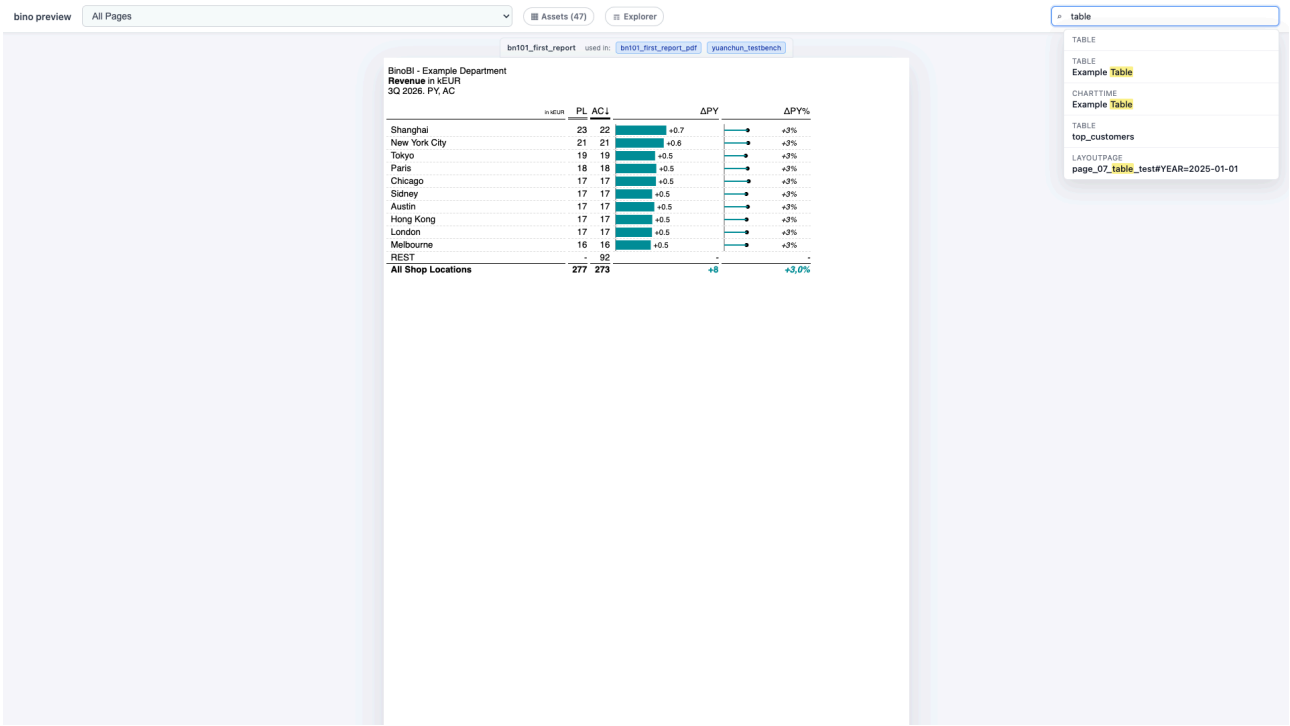


Die Ergebnistabelle zeigt die Spaltennamen mit ihren Datentypen und die Datenzeilen. Am unteren Rand findest du die Seitennavigation, die Anzahl angezeigter Zeilen sowie die Gesamtanzahl der Ergebniszeilen und die Ausführungszeit der Abfrage.

> **Tipp:** Der Data Explorer ist besonders nützlich, um DataSet-Abfragen zu debuggen. Kopiere die SQL-Abfrage aus deinem DataSet-Manifest in den Explorer und prüfe die Ergebnisse, bevor du sie in einer Komponente visualisierst.

14.05 Suchleiste

Am rechten Rand der Toolbar befindet sich die Suchleiste. Sie ermöglicht die schnelle Suche nach Elementen in deinem Projekt:



Wenn du einen Suchbegriff eingibst (z. B. `table`), erscheint ein Dropdown mit allen passenden Treffern. Jeder Treffer zeigt:

- Den **Typ** des Elements (z. B. `TABLE`, `CHARTTIME`, `LAYOUTPAGE`)
- Den **Namen** oder **Titel** des Elements

Der Suchbegriff wird in den Ergebnissen hervorgehoben dargestellt. Ein Klick auf einen Treffer navigiert direkt zur entsprechenden Seite oder Komponente in der Vorschau.

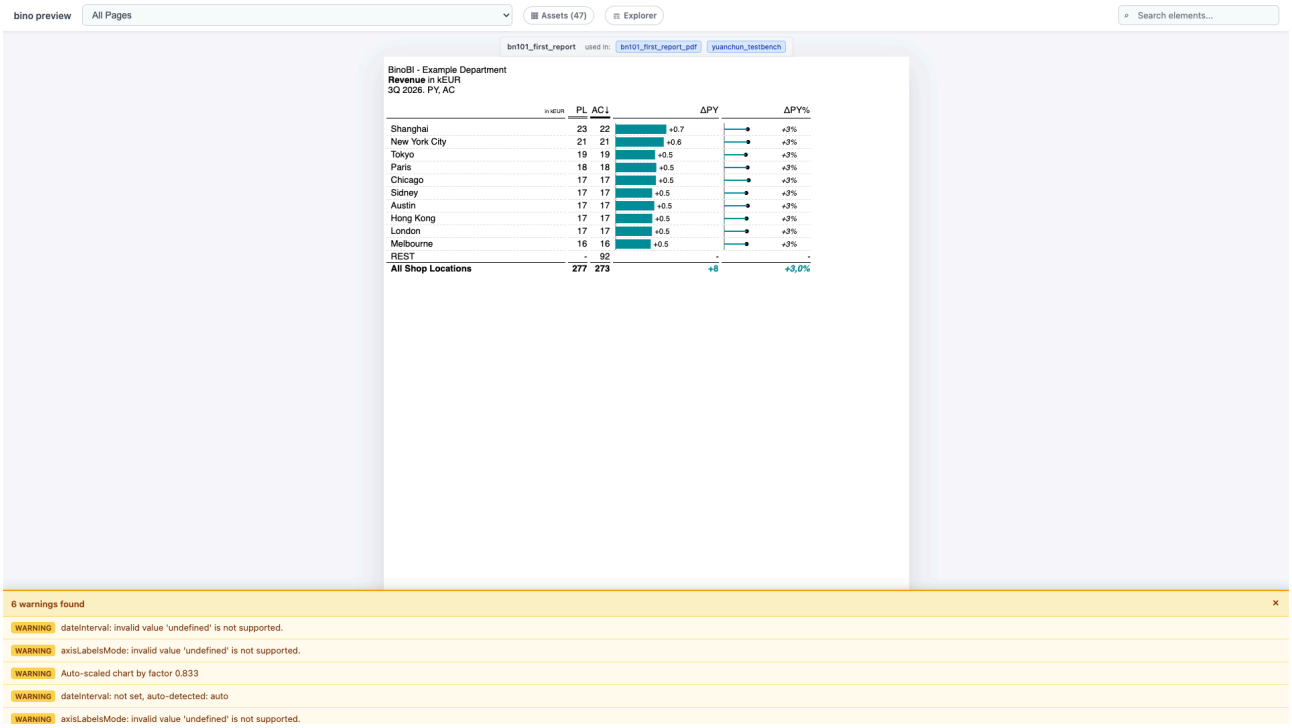
> **Tipp:** Nutze die Suchleiste, um in größeren Projekten schnell zu einer bestimmten Seite oder Komponente zu springen, ohne durch das Artefakt-Dropdown scrollen zu müssen.

14.06 Warning-Panel

Das Warning-Panel erscheint als orangefarbene Leiste am unteren Bildschirmrand. Es zeigt:

- Die **Gesamtzahl** der gefundenen Warnungen (z. B. `6 warnings found`)

- Eine **Liste** aller Warnungen, jeweils mit einem `WARNING`-Badge und einer Beschreibung des Problems



Typische Warnungen umfassen:

Warnungstyp	Beispiel
Ungültige Attributwerte	`dateInterval: invalid value 'undefined' is not supported.`
Automatische Skalierung	`Auto-scaled chart by factor 0.833`
Fehlende Konfiguration	`dateInterval: not set, auto-detected: auto`
Direkte DataSource-Nutzung	`Using datasource(s) "..." directly. Prefer using a dataset`

Warnungen sind oft Hinweise auf Konfigurationsprobleme oder Datenkonstellationen, bei denen die Render-Engine selbstständig Anpassungen vorgenommen hat, um die Seite rendern zu können. Es empfiehlt sich, diese Warnungen zu prüfen und die betreffenden Komponenten explizit zu konfigurieren – auch wenn die implizit angenommenen Werte korrekt sind und übernommen werden sollen.

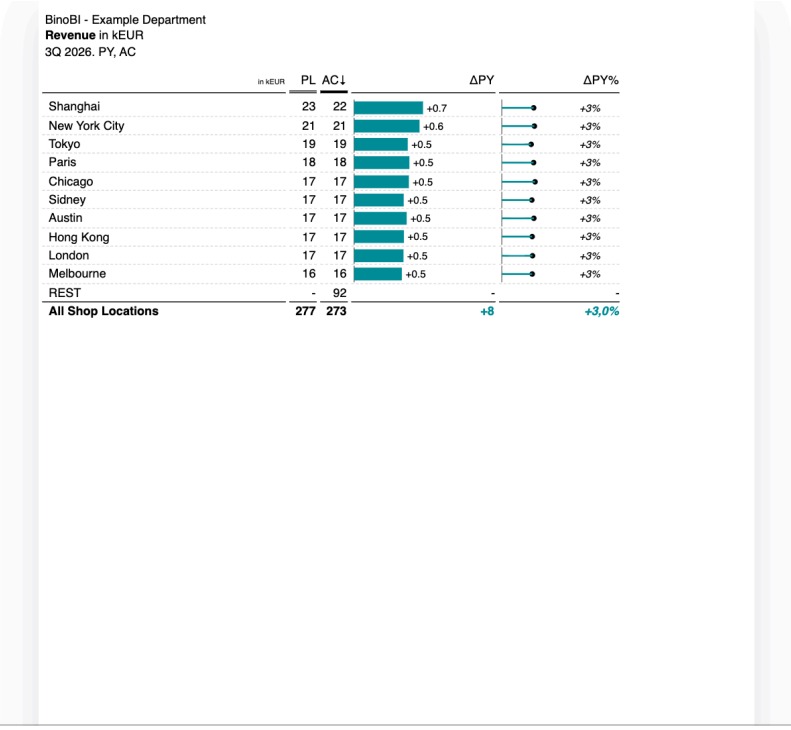
Das Panel lässt sich über das Schließen-Symbol (`x`) oben rechts verbergen. Um es erneut zu öffnen, klicke auf das Warnungs-Badge in der Toolbar.

14.07 Present Mode

Der Present Mode verwandelt einen `ReportArtefact` in eine interaktive Bildschirmpräsentation, basierend auf der Präsentationsbibliothek **Reveal.js**. Jede `LayoutPage` des Artefakts wird dabei zu einer einzelnen Folie.

So aktivierst du den Present Mode:

1. Wähle im Artefakt-Dropdown ein `ReportArtefact` aus (der Present-Button erscheint nur bei Report-Artefakten, nicht bei Document-Artefakten oder der All-Pages-Ansicht).
2. Klicke auf den grünen Button **Present** in der Toolbar.
3. Ein neuer Browser-Tab öffnet sich mit der Präsentation.



Konfiguration über Labels:

Der Present Mode lässt sich über `metadata.labels` am `ReportArtefact` konfigurieren, ohne dass ein separates Manifest nötig ist:

```
apiVersion: bino.bi/v1alpha1
kind: ReportArtefact
metadata:
  name: quarterly-review
  labels:
    presentation.theme: moon
    presentation.transition: fade
    presentation.transitionSpeed: fast
spec:
  format: hd
```

```
  layoutPages:
    - cover
    - highlights
    - summary
```

Die wichtigsten Labels:

Label	Standard	Beschreibung
`presentation.theme`	`white`	Reveal.js-Theme (`black` , `white` , `moon` , `night` , `serif` , `solarized` u. a.)
`presentation.transition`	`slide`	Übergangseffekt (`none` , `fade` , `slide` , `convex` , `concave` , `zoom`)
`presentation.transitionSpeed`	`default`	Geschwindigkeit (`default` , `fast` , `slow`)

Tastenkürzel während der Präsentation:

Taste	Aktion
Pfeiltasten	Navigation zwischen Folien
`Leertaste`	Nächste Folie
`Esc`	Folien-Übersicht
`F`	Vollbildmodus
`B` oder `.`	Pause (schwarzer Bildschirm)

Einzelne Folien konfigurieren:

Über `metadata.labels` auf einzelnen `LayoutPage` -Manifesten kannst du folienspezifische Einstellungen vornehmen:

```
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: cover
  labels:
    presentation.slide.transition: fade
```

> **Tipp:** Derselbe `ReportArtefact` erzeugt sowohl ein PDF (über `bino build`) als auch eine interaktive Präsentation (über den Present-Button). Du brauchst also nicht zwei separate Definitionen für Print und Präsentation.

14.07.01 Beispiel

Für den Present Mode haben wir einen Beispielbericht vorbereitet, den du direkt im Vorschaumodus ausprobieren kannst. Er trägt den Namen `handbuch_example_bn102` – **102 - Preview**.

---

15 Zusammenfassung

In diesem Kapitel hast du die Vorschaufunktion `bino preview` im Detail kennengelernt:

1. **Lokaler Server** – `bino preview` startet einen HTTP-Server auf Port `45678`, der deinen Bericht im Browser rendert.
2. **Hot-Reload** – Änderungen an Manifesten, SQL-Dateien, Datendateien und Assets werden automatisch erkannt und die Ansicht wird neu geladen.
3. **Flags** – Mit `--port`, `--log-sql`, `--lint` und `--data-validation` steuerst du das Verhalten der Vorschau.
4. **bino.toml** – Standardwerte für Flags (`[preview.args]`) und Umgebungsvariablen (`[preview.env]`) lassen sich dauerhaft konfigurieren.
5. **Hooks** – Shell-Befehle an den Checkpoints `pre-preview`, `pre-refresh` und `pre-datasource` automatisieren wiederkehrende Aufgaben.
6. **Constraints** – Mit `mode==preview` bzw. `mode!=build` erstellst du Manifeste, die nur in der Vorschau aktiv sind – ideal für Sampling oder Debug-Ansichten.
7. **Umgebungsvariablen** – Die Vorschau warnt bei nicht aufgelösten Variablen, statt den Build abzubrechen.
8. **Abgrenzung** – `bino preview` ist für die Entwicklung, `bino build` für PDF-Erzeugung und `bino serve` für produktive Web-Anwendungen.
9. **Preview-GUI** – Die Benutzeroberfläche bietet Artefakt-Auswahl, einen Data Explorer für SQL-Abfragen, eine Suchleiste, ein Warning-Panel für Lint-Hinweise und einen Present Mode für Präsentationen.

> **Jetzt bist du dran!** Experimentiere mit den Preview-Optionen: Starte `bino preview --log-sql --lint` und beobachte die Ausgabe im Terminal. Konfiguriere `[preview.args]` in deiner `bino.toml` und ändere ein Manifest – sieh zu, wie die Vorschau sich automatisch aktualisiert. Erstelle ein Debug-DataSet mit `constraints: [mode==preview]` und prüfe,

> ob es nur in der Vorschau erscheint. Öffne den Data Explorer und führe eine SQL-Abfrage auf deinen Daten aus.

---

103 - DataSource und DataSet

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).

In Kapitel 101 hast du eine DataSource und ein DataSet angelegt, um Daten in deinen Bericht zu bringen. Dabei haben wir bewusst nur die Grundlagen gezeigt – ein `SELECT *` als Pass-Through.

Dieses Kapitel erklärt Schritt für Schritt, wie **DataSources** und **DataSets** im Detail funktionieren. Du lernst:

- was eine DataSource ist und welche Typen es gibt,
- wie die Manifest-Struktur einer DataSource aufgebaut ist,
- was ein DataSet ist und wie es auf DataSources aufbaut,
- wie du mit `query`, `prql` und `source` Daten transformierst,
- wie Abhängigkeiten (`dependencies`) die Ausführungsreihenfolge steuern,
- wie das Standard-Datenschema für Visualisierungskomponenten aussieht,
- und wie du externe SQL-Dateien einbindest.

---

01 Das Datenmodell im Überblick

Bevor wir in die Details einsteigen, ist es wichtig, das Zusammenspiel der beiden zentralen Daten-Bausteine zu verstehen:

```
DataSource             DataSet              Komponente
(Rohdaten laden)  -->  (SQL/PRQL)      -->  (Table, Chart, Text ...)
```

- **DataSource** beschreibt *woher* die Daten kommen – z. B. aus einer CSV-Datei, einer Excel-Tabelle, einer Datenbank oder direkt aus dem Manifest (Inline).
- **DataSet** beschreibt *was* mit den Daten geschieht – es transformiert, filtert, aggregiert oder verknüpft die Rohdaten per SQL- oder PRQL-Abfrage.

Jede DataSource wird intern als View in der eingebetteten DuckDB-Engine registriert. DataSets führen darauf SQL-Abfragen aus und stellen ihre Ergebnisse wiederum als View bereit, die von Visualisierungskomponenten (Table, ChartStructure, ChartTime etc.) referenziert werden.

> **Wichtig:** DataSources und DataSets werden nicht in einer festen Reihenfolge verarbeitet. BinoBI baut aus den deklarierten Abhängigkeiten einen **Abhängigkeitsgraphen** (DAG – Directed Acyclic Graph) und stellt so sicher, dass jede Quelle geladen ist, bevor eine darauf aufbauende Abfrage läuft.

> **Wichtig:** BinoBI cached Daten auf Ebene der DataSets. Das bedeutet: Wenn mehrere DataSets auf dieselbe DataSource zugreifen, kann es zu Datenduplikationen kommen. Sollte der verfügbare Speicher für einen Bericht zum Problem werden, kannst du die Anzahl der DataSets reduzieren und stattdessen innerhalb von Visualisierungskomponenten mit Filtern arbeiten.

---

02 DataSource: Struktur und Typen

Eine DataSource ist ein YAML-Manifest vom `kind: DataSource`. Sie beschreibt, wie BinoBI Rohdaten in die Query-Engine lädt. Jede DataSource erzeugt eine View, die nach dem `metadata.name` benannt ist.

02.01 Grundstruktur einer DataSource

```
apiVersion: bino.bi/v1alpha1
kind: DataSource
metadata:
  name: meine_datenquelle
  description: Optionale Beschreibung
spec:
  type: csv
  path: ./resources/data/meine_daten.csv
```

Jedes DataSource-Manifest besteht aus vier Blöcken. Die folgende Tabelle zeigt nicht nur die Hauptblöcke, sondern auch die wichtigsten Unterfelder:

Block	Unterfeld	Typ	Pflicht	Beschreibung
`apiVersion`	-	string	ja	API-Version des Manifests. Für DataSources ist es zum Beispiel `bino.bi/1alpha1`.

Block	Unterfeld	Typ	Pflicht	Beschreibung
`kind`	-	string	ja	Dokumenttyp. Für dieses Manifest immer `DataSource`.
`metadata`	`name`	string	ja	Eindeutiger technischer Name der DataSource. Dieser Name wird später als Tabellenname in SQL verwendet.
`metadata`	`description`	string	nein	Optionale fachliche Beschreibung der DataSource.
`metadata`	`labels`	object	nein	Freie Schlüssel-Wert-Metadaten zur Strukturierung oder Selektion.
`metadata`	`constraints`	string[]	nein	Bedingungen, unter denen das Manifest eingebunden wird, z. B. für bestimmte Modi oder Formate.
`spec`	`type`	string	ja	Typ der Datenquelle, z. B. `csv`, `excel`, `parquet`, `inline`, `postgres_query` oder `mysql_query`.
`spec`	`path`	string	bei Dateiquellen	Pfad zu einer Datei oder ein Glob-Muster, z. B. für CSV-, Excel- oder Parquet-Dateien.
`spec`	`content`	array oder string	bei `inline`	Inline-Daten direkt im Manifest. Alternative zu einer externen Datei.
`spec`	`connection`	object	bei DB-Quellen	Verbindungsdaten für Datenbankquellen wie Host, Port, Datenbank, User und Secret.

Block	Unterfeld	Typ	Pflicht	Beschreibung
`spec`	`query`	string	bei `postgres_query` oder `mysql_query`	SQL-Abfrage, die direkt gegen die externe Datenbank ausgeführt wird.
`spec`	`ephemeral`	boolean	nein	Steuert, ob die Datenquelle zwischen Builds gecacht wird oder bei jedem Lauf neu geladen werden soll.
`spec`	`sample`	number, string, object	nein	Optionales Sampling für große Datenmengen während Entwicklung oder Preview.

> **Hinweis:** Je nach gewähltem DataSource-Typ ändern sich die Pflichtfelder im `spec`-Block.

02.02 Namensregeln

Der `metadata.name` einer DataSource muss dem Muster für SQL-Bezeichner folgen:

- Nur Kleinbuchstaben, Ziffern und Unterstriche: `^[a-z_][a-z0-9_]*$`
- Muss mit einem Buchstaben oder Unterstrich beginnen

Dieser Name wird direkt als Tabellenname in SQL-Abfragen verwendet. Beispiel: Eine DataSource mit `name: umsatz_csv` wird in der Abfrage als `FROM umsatz_csv` referenziert.

02.03 Unterstützte Typen (`spec.type`)

BinoBI unterstützt mehrere Datenquellentypen. Jeder Typ hat eigene Pflichtfelder:

Typ	Beschreibung	Pflichtfelder
`inline`	Daten direkt im YAML-Manifest eingebettet.	`inline` oder `content`
`csv`	CSV-Dateien (einzeln oder per Glob-Muster).	`path`
`excel`	Excel-Dateien (.xlsx).	`path`

Typ	Beschreibung	Pflichtfelder
`parquet`	Parquet-Dateien (einzeln oder per Glob-Muster).	`path`
`postgres_query`	SQL-Abfrage gegen eine PostgreSQL-Datenbank.	`connection`, `query`
`mysql_query`	SQL-Abfrage gegen eine MySQL-Datenbank.	`connection`, `query`

> **Hinweis:** Die Typen `postgres_query` und `mysql_query` sind unter Windows aktuell nicht verfügbar, da die dafür benötigten DuckDB-Erweiterungen auf dieser Plattform noch nicht unterstützt werden.

02.04 Beispiel: Inline-DataSource

Die einfachste Variante – die Daten stehen direkt im Manifest:

```
apiVersion: bino.bi/v1alpha1
kind: DataSource
metadata:
  name: kpi_daten
spec:
  type: inline
  content:
    - { category: "DACH", ac1: 4250, pp1: 3800 }
    - { category: "Nordics", ac1: 2870, pp1: 2650 }
    - { category: "UK", ac1: 1950, pp1: 2100 }
```

Das ist besonders praktisch für kleine, statische Datensätze wie KPI-Definitionen oder Beispieldaten.

02.05 Beispiel: CSV-DataSource

```
apiVersion: bino.bi/v1alpha1
kind: DataSource
metadata:
  name: umsatz_csv
spec:
  type: csv
  path: ./resources/data/umsatz_2026.csv
```

BinoBI erkennt Trennzeichen, Kopfzeile und Datentypen automatisch. Wenn die automatische Erkennung nicht ausreicht, kannst du CSV-Reader-Optionen angeben:

```
apiVersion: bino.bi/v1alpha1
kind: DataSource
metadata:
  name: umsatz_eu
spec:
  type: csv
  path: ./resources/data/umsatz_eu.csv
  delimiter: ";"
  thousands: "."
  decimalSeparator: ","
  dateFormat: "%d/%m/%Y"
```

CSV-Reader-Optionen im Überblick

Feld	Beschreibung	Standard
`delimiter`	Feldtrennzeichen (z. B. `";"` oder `","`).	automatisch
`header`	Erste Zeile als Spaltennamen verwenden.	`true`
`skipRows`	Anzahl der Zeilen, die vor den Daten übersprungen werden.	`0`
`thousands`	Tausendertrennzeichen (z. B. `"."`).	–
`decimalSeparator`	Dezimaltrennzeichen (z. B. `","`).	–
`dateFormat`	Datumsformat (DuckDB-strftime-Syntax, z. B. `"%d/%m/%Y"`).	–
`columnNames`	Explizite Spaltennamen (wenn keine Kopfzeile vorhanden).	–
`columns`	Spaltenname-zu-DuckDB-Typ-Zuordnung (z. B. `amount: "DECIMAL(10,2)"`).	–

> **Tipp:** `columnNames` und `columns` sind gegenseitig exklusiv – verwende entweder das eine oder das andere.

02.06 Beispiel: Excel-DataSource

```
apiVersion: bino.bi/v1alpha1
kind: DataSource
metadata:
  name: budget_excel
spec:
  type: excel
  path: ./resources/data/budget_2026.xlsx
```

02.07 Beispiel: Parquet-DataSource

Parquet eignet sich besonders für große Datensätze. Du kannst auch Glob-Muster verwenden:

```
apiVersion: bino.bi/v1alpha1
kind: DataSource
metadata:
  name: fact_sales
spec:
  type: parquet
  path: ./warehouse/fact_sales/*.parquet
```

02.08 Beispiel: PostgreSQL-DataSource

Für Datenbankverbindungen benötigst du zusätzlich ein `ConnectionSecret`-Manifest, das die Zugangsdaten verwaltet:

```
---
apiVersion: bino.bi/v1alpha1
kind: ConnectionSecret
metadata:
  name: pg_credentials
spec:
  type: postgres
  postgres:
    passwordFromEnv: POSTGRES_PASSWORD
---
apiVersion: bino.bi/v1alpha1
kind: DataSource
metadata:
  name: umsatz_pg
spec:
```

```
  type: postgres_query
  connection:
    host: ${DB_HOST:db.example.com}
    port: 5432
    database: analytics
    schema: public
    user: reporting
    secret: pg_credentials
  query: |
    SELECT region, booking_date, amount
    FROM fact_sales
    WHERE booking_date >= DATE '2026-01-01'
```

> **Hinweis:** Das Passwort wird über eine Umgebungsvariable bereitgestellt (`POSTGRES_PASSWORD`), nicht im Manifest gespeichert. Setze die Variable vor dem Build: `export POSTGRES_PASSWORD='geheim'` . Details zu ConnectionSecrets findest du in der Referenz (https://cli.bino.bi/reference/connection-secret/).

02.09 Caching mit `ephemeral`

Das Feld `ephemeral` steuert, ob BinoBI die Daten einer DataSource zwischen Builds zwischenspeichert:

- **Nicht gesetzt** (Standard): BinoBI leitet das Verhalten automatisch ab. Dateibasierte Quellen (CSV, Parquet) werden gecacht; Datenbankabfragen werden als veränderlich behandelt.
- **`ephemeral: false`** : Caching wird explizit erlaubt – nützlich für Datenbankabfragen, deren Ergebnis sich zwischen Builds nicht ändert.
- **`ephemeral: true`** : Daten werden bei jedem Build neu geladen.

02.10 Sampling – große Datenmengen während der Entwicklung begrenzen

Mit dem `sample` -Feld kannst du während der Entwicklung nur einen Teil der Daten laden, um schneller iterieren zu können:

```
spec:
  type: parquet
  path: ./warehouse/fact_sales/*.parquet
  sample: 5000 # Nur 5000 Zeilen laden
```

Drei Formen sind möglich:

Form	Beispiel	Beschreibung
Zahl	`sample: 5000`	Genau N Zeilen laden.
String	`sample: "10%"`	Ungefähr N % der Zeilen laden.
Objekt	`sample: { size: 5000, method: reservoir }`	Exakte Anzahl mit gewählter Methode.

> **Tipp:** Kombiniere `sample` mit Constraints (siehe Kapitel 105), um während der Vorschau gesampelte Daten zu verwenden und im produktiven Build den vollständigen Datensatz.

02.11 DataSources erweitern mit Plugins

Die eingebauten DataSource-Typen (`inline` , `csv` , `excel` , `parquet` , `postgres_query` , `mysql_query`) decken die gängigsten Anwendungsfälle ab. Darüber hinaus bietet BinoBI eine **Plugin-Engine**, mit der du eigene DataSource-Typen erstellen oder bestehende erweitern kannst.

Ein Plugin kann beispielsweise Daten aus SaaS-APIs, proprietären Datenbanken oder beliebigen anderen Quellen laden. Die Daten fließen dabei genauso in die DuckDB-Engine wie bei den eingebauten Typen – für DataSets und Komponenten ist der Ursprung transparent.

```
# Beispiel: Plugin-basierte DataSource (erfordert installiertes Plugin)
apiVersion: bino.bi/v1alpha1
kind: SalesforceDataSource
metadata:
  name: opportunities
spec:
  object: Opportunity
  query: "SELECT Id, Name, Amount FROM Opportunity WHERE StageName = 'Closed Wor
```

> **Hinweis:** Die Erstellung und Konfiguration von Plugins wird in einem späteren Kapitel ausführlich behandelt. Weitere Informationen findest du vorab in der Plugin-Dokumentation (https://cli.bino.bi/guides/plugins/).

02.12 Attribut-Referenz DataSource

Die folgende Tabelle fasst die wichtigsten `spec` -Attribute zusammen. Die vollständige Referenz findest du unter cli.bino.bi/reference/datasource/ (https://cli.bino.bi/reference/datasource/).

Attribut	Typ	Pflicht	Beschreibung
`spec.type`	string	ja	Datenquellentyp: `inline`, `csv`, `excel`, `parquet`, `postgres_query`, `mysql_query`.
`spec.path`	string	bei Dateitypen	Dateipfad oder Glob-Muster.
`spec.content`	array/string	bei `inline`	Inline-Daten als Array oder JSON-String.
`spec.connection`	object	bei DB-Typen	Datenbankverbindungsdaten (Host, Port, Datenbank, User, Secret).
`spec.query`	string	bei DB-Typen	SQL-Abfrage an die Datenbank.
`spec.ephemeral`	boolean	nein	Caching-Steuerung.
`spec.sample`	number/string/object	nein	Zeilenbegrenzung für Sampling.
`spec.delimiter`	string	nein	CSV-Feldtrennzeichen.
`spec.header`	boolean	nein	Erste Zeile als Kopfzeile (CSV). Standard: `true`.
`spec.skipRows`	integer	nein	Zeilen vor den Daten überspringen (CSV).
`spec.thousands`	string	nein	Tausendertrennzeichen (CSV).
`spec.decimalSeparator`	string	nein	Dezimaltrennzeichen (CSV).
`spec.dateFormat`	string	nein	Datumsformat (CSV, DuckDB-strftime).
`spec.columnNames`	string[]	nein	Explizite Spaltennamen (CSV). Exklusiv mit `columns`.
`spec.columns`	object	nein	Spaltenname-zu-Typ-Zuordnung (CSV). Exklusiv mit `columnNames`.

03 DataSet: Daten transformieren mit SQL

Ein DataSet ist ein YAML-Manifest vom `kind: DataSet`. Es führt eine SQL- oder PRQL-Abfrage auf den in DataSources (oder anderen DataSets) geladenen Daten aus und stellt das Ergebnis als benannte Tabelle bereit.

03.01 Grundstruktur eines DataSets

```
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: umsatz_nach_region
  description: Umsatz aggregiert nach Region
spec:
  query: |
    SELECT
      region AS category,
      SUM(amount) AS ac1
    FROM umsatz_csv
    GROUP BY region
  dependencies:
    - umsatz_csv
```

Ein DataSet-Manifest hat drei zentrale `spec`-Felder, von denen genau eines gesetzt sein muss:

Feld	Beschreibung
`spec.query`	SQL-Abfrage (inline oder aus externer Datei). Die gängigste Variante.
`spec.prql`	PRQL-Abfrage (inline oder aus externer Datei). Alternative zu SQL.
`spec.source`	Direktes Pass-Through einer DataSource ohne Transformation. Alternative zu `query` und `prql`.

Zusätzlich:

Feld	Beschreibung
`spec.dependencies`	Liste der DataSources oder DataSets, die in der Abfrage referenziert werden.

03.02 Tabellenname in der Abfrage

In der `FROM`-Klausel deiner SQL-Abfrage verwendest du den **`metadata.name`** der DataSource (oder eines anderen DataSets) als Tabellennamen:

```
SELECT * FROM umsatz_csv
```

> **Achtung:** Der Tabellenname muss exakt dem `metadata.name` der referenzierten DataSource oder des referenzierten DataSets entsprechen – nicht dem Dateinamen der YAML-Datei und nicht dem Dateinamen der CSV-Datei.

---

04 `query` - die SQL-Abfrage im Detail

Das Feld `query` enthält eine SQL-Abfrage, die von der eingebetteten **DuckDB**-Engine ausgeführt wird. Du kannst den vollen SQL-Funktionsumfang von DuckDB nutzen – das umfasst deutlich mehr als nur `SELECT *`.

04.01 Unterstützter SQL-Umfang

Feature	Beispiel	Beschreibung
Spaltenauswahl	`SELECT category, ac1, pp1 FROM quelle`	Nur bestimmte Spalten übernehmen.
Aliase	`SELECT category AS region FROM quelle`	Spalten umbenennen.
Berechnete Spalten	`SELECT *, ac1 - pp1 AS diff FROM quelle`	Neue Spalten aus Berechnungen ableiten.
Filter	`SELECT * FROM quelle WHERE ac1 > 1000`	Zeilen nach Bedingungen einschränken.
Aggregation	`SELECT category, SUM(ac1) AS total FROM quelle GROUP BY category`	Werte gruppieren und aggregieren (`SUM`, `AVG`, `COUNT`, `MIN`, `MAX` etc.).
Sortierung	`SELECT * FROM quelle ORDER BY ac1 DESC`	Ergebnisse sortieren.
Limit	`SELECT * FROM quelle LIMIT 10`	Nur eine bestimmte Anzahl Zeilen zurückgeben.

Feature	Beispiel	Beschreibung
Joins	`SELECT a.*, b.name FROM quelle_a a JOIN quelle_b b ON a.id = b.id`	Mehrere Quellen verknüpfen (alle JOIN-Typen: `INNER`, `LEFT`, `RIGHT`, `FULL`).
Subqueries	`SELECT * FROM (SELECT category, SUM(ac1) AS s FROM quelle GROUP BY category) sub`	Verschachtelte Abfragen als Unterabfragen.
Window Functions	`SELECT *, RANK() OVER (ORDER BY ac1 DESC) AS rang FROM quelle`	Analytische Funktionen wie `RANK`, `ROW_NUMBER`, `LAG`, `LEAD`.
CTEs	`WITH top AS (SELECT * FROM quelle WHERE ac1 > 500) SELECT * FROM top`	Common Table Expressions für lesbarere Abfragen.
CASE-Ausdrücke	`SELECT *, CASE WHEN ac1 > 1000 THEN 'hoch' ELSE 'niedrig' END AS klasse FROM quelle`	Bedingte Logik innerhalb der Abfrage.
String-Funktionen	`SELECT UPPER(category) AS cat FROM quelle`	Textmanipulation (`UPPER`, `LOWER`, `CONCAT`, `REPLACE` etc.).
Typ-Konvertierung	`SELECT CAST(ac1 AS INTEGER) FROM quelle`	Datentypen umwandeln.
UNION / INTERSECT	`SELECT * FROM quelle_a UNION ALL SELECT * FROM quelle_b`	Ergebnisse mehrerer Abfragen kombinieren.

04.02 Pass-Through vs. eigene Query

Wenn du **keine Transformation** benötigst, kannst du statt `query` auch das Kurzform-Feld `source` verwenden:

```
spec:
  source: umsatz_csv
```

Das ist funktional identisch zu `query: "SELECT * FROM umsatz_csv"`, spart aber Tipparbeit. Verwende `source`, wenn du die Daten unverändert weiterreichen möchtest, und `query`, sobald du filtern, aggregieren oder transformieren willst.

04.03 Externe SQL-Dateien

Für komplexere Abfragen kannst du die SQL-Anweisung in eine externe Datei auslagern und im Manifest über die `$file`-Syntax referenzieren. Das hält die YAML-Dateien übersichtlich und ermöglicht Syntax-Highlighting im Editor.

```
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: umsatz_nach_region
spec:
  query:
    $file: ./resources/sql/umsatz_nach_region.sql
  dependencies:
    - umsatz_csv
```

Die Datei `resources/sql/umsatz_nach_region.sql` enthält dann die eigentliche SQL-Abfrage:

```
SELECT
  region AS category,
  SUM(amount) AS ac1,
  SUM(budget) AS pl1
FROM umsatz_csv
GROUP BY region
ORDER BY ac1 DESC
```

Pfadauflösung:

- Pfade werden relativ zur Manifest-Datei aufgelöst, die den `$file`-Verweis enthält.
- Sowohl `./relativ/pfad.sql` als auch `relativ/pfad.sql` sind möglich.

Auto-Reload: Externe SQL-Dateien sind vollständig in Caching und Hot-Reload integriert. Änderungen an der SQL-Datei führen automatisch zu einem Neu-Build des DataSets und werden in `bino preview` sofort sichtbar.

Empfohlene Verzeichnisstruktur für externe Queries:

```
mein-bericht/
+-- bino.toml
+-- datasets/
|   +-- umsatz.yaml
```

```
+-- resources/
    +-- sql/
        +-- umsatz_nach_region.sql
        +-- umsatz_monatlich.sql
```

04.04 PRQL als Alternative zu SQL

Neben SQL unterstützt BinoBI auch PRQL (https://prql-lang.org) (Pipelined Relational Query Language) – eine moderne, pipeline-basierte Abfragesprache, die zu SQL kompiliert wird.

```
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: umsatz_nach_region
spec:
  prql: |
    from umsatz_csv
    filter amount > 0
    group {region} (
      aggregate {
        total = sum amount
      }
    )
    sort {-total}
  dependencies:
    - umsatz_csv
```

PRQL kann ebenfalls aus externen Dateien geladen werden:

```
spec:
  prql:
    $file: ./resources/sql/umsatz_nach_region.prql
```

> **Hinweis:** Wenn `spec.prql` gesetzt ist, hat es Vorrang vor `spec.query`. Die Felder `query`, `prql` und `source` sind gegenseitig exklusiv – setze immer nur eines davon.

---

05 `dependencies` - die Abhängigkeiten

Das Feld `dependencies` ist eine Liste von Namen, die angibt, welche **DataSources** oder **anderen DataSets** in der `query` referenziert werden. Es erfüllt zwei zentrale Aufgaben:

05.01 Ausführungsreihenfolge sicherstellen

BinoBI verarbeitet DataSources und DataSets nicht in einer festen Reihenfolge, sondern baut aus den `dependencies` einen **Abhängigkeitsgraphen** (DAG – Directed Acyclic Graph). Dadurch wird garantiert, dass eine Quelle oder ein vorgelagertes DataSet vollständig geladen ist, *bevor* die darauf aufbauende Abfrage ausgeführt wird.

Beispiel: Wenn DataSet `B` von DataSet `A` abhängt und DataSet `A` von DataSource `csv_quelle`, dann ist die Ausführungsreihenfolge:

```
csv_quelle  -->  DataSet A  -->  DataSet B
```

Du kannst den Abhängigkeitsgraphen deines Projekts visualisieren mit:

```
bino graph
```

05.02 Tabellen in der Query-Engine registrieren

Nur Quellen und DataSets, die in `dependencies` aufgeführt sind, werden als benannte Tabellen in der DuckDB-Engine registriert und stehen damit in der `FROM`-Klausel zur Verfügung.

> **Wichtig:** Führe jeden Tabellennamen, den du in der `query` verwendest, auch in `dependencies` auf. `dependencies` bestimmt die Ausführungsreihenfolge im DAG und macht die Abhängigkeit in `bino graph` sichtbar.
>
> Verlass dich aber nicht darauf, dass ein fehlender Eintrag auffällt: DataSources sind ohnehin registriert, eine Query auf eine nicht deklarierte DataSource läuft also stillschweigend durch. Erst bei einem DataSet, das vor der Ausführung selbst berechnet werden muss, führt der fehlende Eintrag zu einem Fehler.

05.03 Beispiel mit mehreren Abhängigkeiten (Join)

```
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: umsatz_mit_wechselkurs
spec:
  query: |
```

```
    SELECT
      u.region AS category,
      u.amount * w.rate_to_eur AS ac1
    FROM umsatz_csv u
    JOIN wechselkurse w ON u.currency = w.currency
  dependencies:
    - umsatz_csv
    - wechselkurse
```

Hier werden zwei DataSources als Abhängigkeiten deklariert, sodass beide als Tabellen verfügbar sind und per `JOIN` verknüpft werden können.

05.04 Wann kann `dependencies` weggelassen werden?

Wenn du das Kurzform-Feld `source` statt `query` verwendest, ist `dependencies` optional – BinoBI leitet die Abhängigkeit automatisch aus dem `source`-Wert ab:

```
spec:
  source: umsatz_csv
  # dependencies wird automatisch als [umsatz_csv] angenommen
```

---

06 DataSet-Verkettung: DataSets auf DataSets

Ein DataSet kann nicht nur auf DataSources zugreifen, sondern auch auf andere DataSets. So kannst du komplexe Transformationen in mehrere, übersichtliche Schritte zerlegen.

```
# Schritt 1: Rohdaten aggregieren
---
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: umsatz_aggregiert
spec:
  query: |
    SELECT region, SUM(amount) AS total
    FROM umsatz_csv
    GROUP BY region
  dependencies:
    - umsatz_csv
```

```
# Schritt 2: Auf dem Zwischenergebnis weiterarbeiten
---
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: umsatz_top5
spec:
  query: |
    SELECT region AS category, total AS ac1
    FROM umsatz_aggregiert
    ORDER BY total DESC
    LIMIT 5
  dependencies:
    - umsatz_aggregiert
```

Die Ausführungsreihenfolge wird automatisch sichergestellt:

```
umsatz_csv  -->  umsatz_aggregiert  -->  umsatz_top5
```

---

07 Inline-DataSets und Inline-DataSources

Für einfache, komponentenspezifische Daten kannst du DataSets und DataSources **inline** direkt in einer Komponente definieren, ohne separate Manifest-Dokumente zu erstellen.

07.01 Inline-DataSet in einer Komponente

```
apiVersion: bino.bi/v1alpha1
kind: ChartStructure
metadata:
  name: umsatz_chart
spec:
  dataset:
    query: |
      SELECT region AS category, SUM(amount) AS ac1
      FROM umsatz_csv
      GROUP BY region
```

```
    dependencies:
      - umsatz_csv
  chartTitle: Umsatz nach Region
  chartType: bar
```

07.02 Inline-DataSet mit Inline-DataSource

Du kannst sogar die DataSource selbst inline innerhalb der `dependencies` definieren. Die Referenzierung erfolgt über die `@inline(N)`-Syntax, wobei `N` der Index (0-basiert) im `dependencies`-Array ist:

```
apiVersion: bino.bi/v1alpha1
kind: ChartStructure
metadata:
  name: schnell_chart
spec:
  dataset:
    dependencies:
      - type: csv
        path: ./resources/data/umsatz.csv
    query: |
      SELECT region AS category, SUM(amount) AS ac1
      FROM @inline(0)
      GROUP BY region
  chartTitle: Schnellansicht
```

07.03 Benannte und Inline-Referenzen mischen

Du kannst benannte DataSource-Referenzen und Inline-Definitionen im selben `dependencies`-Array kombinieren:

```
spec:
  dataset:
    dependencies:
      - umsatz_csv # Benannte Referenz auf existierende DataSource
      - type: csv # Inline-Definition
        path: ./resources/data/wechselkurse.csv
    query: |
      SELECT u.*, w.rate
```

```
    FROM umsatz_csv u
    JOIN @inline(1) w ON u.currency = w.code
```

Benannte Referenzen verwenden ihren Dokumentnamen direkt in SQL, während Inline-Definitionen über `@inline(N)` referenziert werden.

> **Hinweis:** Benutzerdefinierte Dokumentnamen dürfen **nicht** mit `_inline_` beginnen – dieses Präfix ist für generierte Inline-Definitionen reserviert.

---

08 Das Standard-Datenschema

Damit Visualisierungskomponenten wie Charts und Tabellen korrekt funktionieren, sollten DataSet-Abfragen Zeilen liefern, die dem **Standard-Datenschema** entsprechen. Nicht alle Felder sind für jede Komponente erforderlich, aber die Einhaltung der Struktur stellt sicher, dass Aggregation, Drill-down und Filterung korrekt funktionieren.

08.01 Messwerte (Szenarien)

Das Schema unterstützt vier parallele Szenario-Sätze für vergleichende Analysen:

Spalte	Beschreibung
`ac1` … `ac4`	**Actual** – Ist-Werte (aktuelle Messwerte).
`pp1` … `pp4`	**Previous Period** – Vorperiode (Vorjahr, Vormonat).
`fc1` … `fc4`	**Forecast** – Prognosewerte.
`pl1` … `pl4`	**Plan/Budget** – Ziel- oder Planwerte.

08.02 Gruppierungsdimensionen

Diese Felder bestimmen, wie Daten hierarchisch organisiert und sortiert werden:

Dimension	Index-Spalte	Beschreibung
`rowGroup`	`rowGroupIndex`	Übergeordnete Zeilengruppierung (z. B. “Umsatz”, “Kosten”).
`category`	`categoryIndex`	Primäre Datendimension (z. B. “Produkt A”, “Region Nord”).

Dimension	Index-Spalte	Beschreibung
`subCategory`	`subCategoryIndex`	Detail-Dimension für Drill-down (erzeugt "davon"-Zeilen).
`columnGroup`	`columnGroupIndex`	Spaltengruppierung für die Aufgliederung von Messwerten.

> **Hinweis:** Wenn du eine Gruppierungsdimension (z. B. `category`) angibst, solltest du auch die zugehörige Index-Spalte (z. B. `categoryIndex`) liefern, um eine konsistente Sortierung sicherzustellen.

08.03 Metadaten- und Steuerfelder

Spalte	Typ	Beschreibung
`date`	string	**Pflicht bei ChartTime.** ISO-8601-Datum (z. B. `2024-01-15`). Für Zeitachsen.
`operation`	string	Aggregationsvorzeichen: `'+'` (addieren) oder `'-'` (subtrahieren). Standard: `'+'`.
`setname`	string	DataSet-Bezeichner dient als zusätzliche Filtermöglichkeit, wenn Daten aus unterschiedlichen Quellen zusammen gestellt werden.

08.04 Beispielabfrage für in standardkonformes DataSet

```
SELECT
  'Umsatz' AS rowGroup,
  1 AS rowGroupIndex,
  product_name AS category,
  product_rank AS categoryIndex,
  '2026-01-15' AS date,
  SUM(sales_amount) AS ac1,
  SUM(budget_amount) AS pl1
FROM umsatz_csv
GROUP BY product_name, product_rank
```

---

09 Datenvalidierung

BinoBI kann Abfrageergebnisse zur Build-Zeit gegen das Standard-Datenschema validieren. Das hilft, häufige Datenfehler früh zu erkennen.

09.01 Was wird geprüft?

Prüfung	Beschreibung
Typvalidierung	String-Felder (`category` , `rowGroup` etc.) müssen Strings sein.
Typvalidierung	Zahlenfelder (`ac1` , `categoryIndex` etc.) müssen Zahlen sein.
Enum-Validierung	`operation` muss `"+"` oder `"−"` sein.
Datumsformat	`date` muss im ISO-8601-Format vorliegen (`YYYY-MM-DD`).
Abhängige Pflichtfelder	`rowGroup` erfordert `rowGroupIndex` , `category` erfordert `categoryIndex` etc.

09.02 Validierung konfigurieren

Die Datenvalidierung ist standardmäßig im `warn` -Modus aktiv. Du kannst das Verhalten über CLI-Flags steuern:

```
# Warnungen ausgeben und fortfahren (Standard)
bino build --data-validation=warn

# Validierungsfehler als fatal behandeln
bino build --data-validation=fail

# Validierung komplett deaktivieren
bino build --data-validation=off
```

Modus	Wirkung
`warn`	Verletzungen des Datenschemas werden protokolliert, der Build läuft aber weiter und erzeugt das Artefakt. Das ist der Standard.
`fail`	Eine Schemaverletzung bricht den Lauf mit einem Fehler ab; es wird kein Artefakt erzeugt. Sinnvoll in CI/CD-Pipelines.
`off`	Die Validierung wird komplett übersprungen. Schnellster Lauf, aber Datenfehler bleiben unentdeckt.

Für eine reine Validierung ohne Build verwende:

```
bino lint --execute-queries
```

09.03 Stichprobengröße

Aus Effizienzgründen validiert BinoBI standardmäßig nur die ersten 10 Zeilen pro DataSet. Die Größe lässt sich über eine Umgebungsvariable anpassen:

```
BNR_DATA_VALIDATION_SAMPLE_SIZE=100 bino build
```

---

10 Constraints: DataSets bedingt einbinden

DataSource- und DataSet-Dokumente unterstützen `metadata.constraints`, um sie bedingt für bestimmte Artefakte, Modi oder Ausgabeformate einzubinden – z. B. Debug-DataSets nur in der Vorschau oder format-spezifische Aggregationen für Print vs. Screen.

> **Hinweis:** Eine ausführliche Erklärung des Constraint-Systems mit allen Operatoren, Feldern und Beispielen findest du in Kapitel 105.

---

11 Vollständiges Beispiel: Von der Quelle zum Chart

Das folgende Beispiel zeigt den kompletten Weg von der DataSource über das DataSet bis zur Visualisierung:

```
# 1. DataSource: CSV-Datei als Rohdaten
---
apiVersion: bino.bi/v1alpha1
kind: DataSource
metadata:
  name: umsatz_csv
spec:
  type: csv
  path: ./resources/data/umsatz_2026.csv
  delimiter: ";"
  decimalSeparator: ","
```

```
# 2. DataSet: Daten transformieren
---
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: umsatz_nach_region
spec:
  query: |
    SELECT
      region AS category,
      ROW_NUMBER() OVER (ORDER BY SUM(amount) DESC) AS categoryIndex,
      SUM(amount) AS ac1,
      SUM(budget) AS pl1,
      SUM(prev_year) AS pp1
    FROM umsatz_csv
    GROUP BY region
  dependencies:
    - umsatz_csv


# 3. LayoutPage: Daten visualisieren
---
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: seite_umsatz
spec:
  pageFormat: a4
  pageOrientation: landscape
  pageLayout: split-vertical
  titleBusinessUnit: "BinoBI - Vertrieb"
  titleDateStart: 2026-01-01
  titleDateEnd: 2026-12-31
  titleMeasures:
    - name: "Umsatz"
      unit: "kEUR"
  titleScenarios: [ac1, pp1]
  titleDateFormat: year
  titleDateLink: end
  children:
    - kind: Table
```

```
    spec:
      dataset: umsatz_nach_region
      sumTitle: Umsatz nach Region
      scenarios: [ac1, pl1]
      variances: [dac1_pp1_pos]
      barColumns: [dac1_pp1_pos]
      type: sum
      measureScale: k
      measureUnit: EUR
  - kind: ChartStructure
    spec:
      dataset: umsatz_nach_region
      chartTitle: Top Regionen
      scenarios: [ac1, pp1]
      variances: [dac1_pp1_pos]
      measureScale: k
      measureUnit: EUR
```

---

12 Zusammenfassung

In diesem Kapitel hast du die beiden zentralen Daten-Bausteine von BinoBI im Detail kennengelernt:

DataSource:

1. Beschreibt *woher* Daten kommen (CSV, Excel, Parquet, Datenbank, Inline).
2. Erzeugt eine benannte View in der DuckDB-Engine.
3. Unterstützt CSV-Reader-Optionen, Sampling und Caching-Steuerung.
4. Kann über die **Plugin-Engine** um eigene Typen erweitert werden – Details dazu folgen in einem späteren Kapitel.

DataSet:

1. Beschreibt *was* mit den Daten geschieht (SQL, PRQL oder Pass-Through).
2. Deklariert Abhängigkeiten (`dependencies`), die die Ausführungsreihenfolge steuern.
3. Kann auf DataSources und andere DataSets zugreifen (Verkettung).
4. Unterstützt externe SQL-/PRQL-Dateien über die `$file` -Syntax.
5. Kann als Inline-DataSet direkt in einer Komponente definiert werden, ohne ein eigenes Manifest anzulegen (Abschnitt 07).

6. Sollte dem Standard-Datenschema folgen (Szenarien, Dimensionen, Metadaten).

Pipeline:

```
DataSource  -->  DataSet  -->     Komponente  -->  LayoutPage  -->  ReportArtef
(Laden)        (Transformieren)  (Darstellen)      (Anordnen)      (Zusammenbaue
```

> **Jetzt bist du dran!** Experimentiere mit eigenen SQL-Abfragen: Erstelle ein DataSet mit Aggregation, verknüpfe zwei DataSources per JOIN oder lagere eine komplexe Abfrage in eine externe SQL-Datei aus. Starte `bino preview --log-sql` und beobachte, welche SQL-Abfragen BinoBI ausführt. Ändere die Abfrage, speichere und sieh das Ergebnis sofort im Browser.

104 – Seitenlayout und LayoutCards

> **Disclaimer:** Die in diesem Kapitel vorgestellten Component-Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).

In Kapitel 101 hast du eine einfache Berichtsseite mit einer einzelnen Tabelle erstellt. In der Praxis sollen Berichte aber oft mehrere Visualisierungen nebeneinander zeigen – zum Beispiel eine Tabelle links und ein Diagramm rechts oder vier Kennzahlenblöcke in einem 2×2-Raster.

Dieses Kapitel zeigt dir, wie du mit **Seitenlayouts** (`pageLayout`) und **LayoutCards** die Anordnung von Komponenten auf einer Seite steuerst. Du lernst:

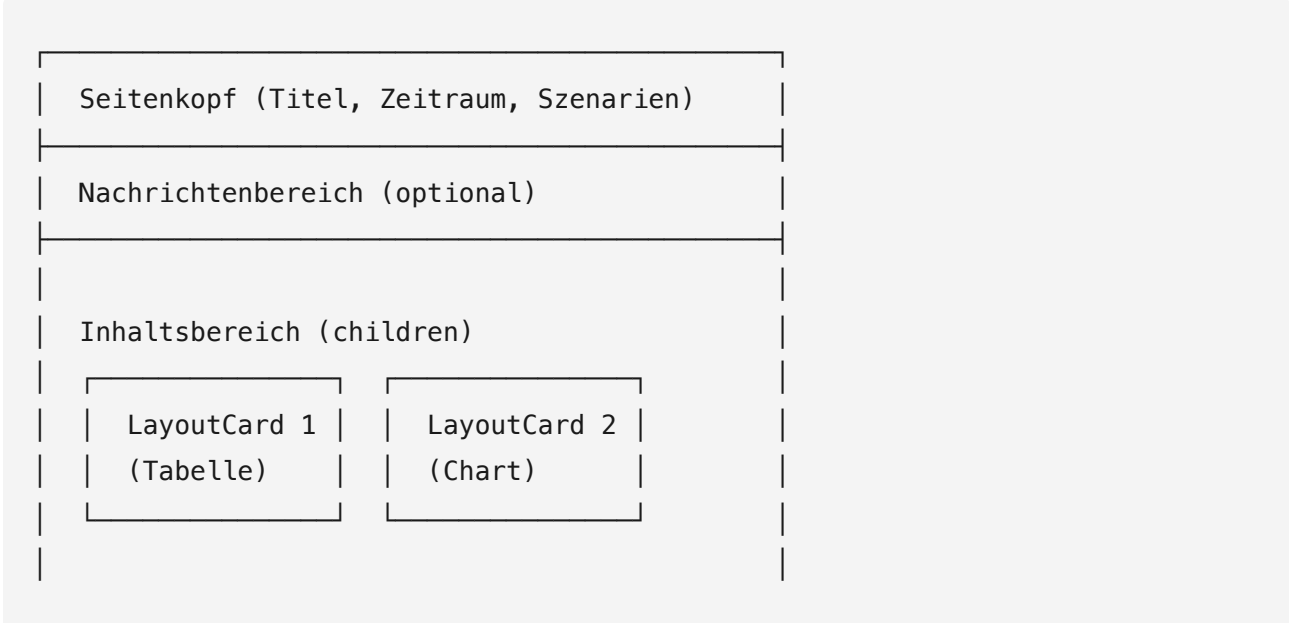
- wie du den verfügbaren Platz auf einer Seite aufteilst,
- wie LayoutCards als eigenständige Inhaltsblöcke mit eigenem Titel funktionieren,
- wie du den Seitenkopf, die Fußzeile und den Nachrichtenbereich konfigurierst,
- und wie du Seiten für verschiedene Formate und Ausrichtungen optimierst.

---

01 – Die LayoutPage im Überblick

Jede Berichtsseite ist ein `LayoutPage` -Manifest. Es beschreibt nicht nur *welche* Komponenten angezeigt werden (`children`), sondern auch *wie* sie auf der Seite angeordnet sind.

Eine LayoutPage besteht aus mehreren Bereichen:



```
├──────────────────────────────────────────────┤
│  Fußzeile (optional)                         │
└──────────────────────────────────────────────┘
```

Layout 1: Layout Überblick

01.01 Grundgerüst einer LayoutPage

```
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: meine_seite
  description: Beispielseite mit Layout
spec:
  pageFormat: a4
  pageOrientation: landscape
  pageLayout: split-vertical
  children:
    - kind: Table
      spec:
        dataset: mein_dataset
    - kind: ChartStructure
      spec:
        dataset: mein_dataset
        chartTitle: Rechte Hälfte
```

---

02 – Seitenformat und Ausrichtung

Bevor du dich um die Anordnung der Inhalte kümmerst, legst du das **Format** und die **Ausrichtung** der Seite fest.

02.01 `pageFormat` – das Seitenformat

In der folgenden Tabelle findest du einige beispielhafte unterstützte Seitenformate. Eine vollständige Übersicht aller verfügbaren Optionen findest du in der CLI-Referenz (https://cli.bino.bi/reference/layout-page/).

Wert	Beschreibung	Typischer Einsatz
`a4`	DIN A4 (210 × 297 mm)	Klassische Druckberichte
`letter`	US Letter (216 × 279 mm)	Nordamerikanische Berichte
`xga`	XGA-Bildschirmformat (1024 × 768 px)	Bildschirmpräsentationen, Dashboards
`full-hd`	Full HD (1920 × 1080 px)	HD-Bildschirmexporte, Screenshots
`4k`	4K	4k-Bildschirmexporte, Screenshots

02.02 `pageOrientation` – die Ausrichtung

Wert	Beschreibung
`portrait`	Hochformat – die Seite ist höher als breit.
`landscape`	Querformat – die Seite ist breiter als hoch.

> **Tipp:** Für Dashboards und Präsentationen eignet sich `full-hd` im Querformat besonders gut. Für klassische Druckberichte empfiehlt sich `a4` im Hochformat.

---

03 – Der Seitenkopf

Der Seitenkopf zeigt den übergeordneten Kontext der Seite: Organisationseinheit, Berichtszeitraum, Kennzahlen und Szenarien. Er wird über Felder direkt im `spec`-Block der LayoutPage konfiguriert.

03.01. Alle Titelfelder im Überblick

Feld	Beschreibung	Beispiel
`titleBusinessUnit`	Organisationseinheit im Seitentitel.	`"BinoBI – Example Department"`
`titleDateStart`	Startdatum des Berichtszeitraums (ISO 8601).	`2026-01-01`
`titleDateEnd`	Enddatum des Berichtszeitraums (ISO 8601).	`2026-09-30`

Feld	Beschreibung	Beispiel
`titleMeasures`	Kennzahlen mit Name und Einheit.	`[{name: "Revenue", unit: "kEUR"}]`
`titleScenarios`	Szenarien-Kürzel im Titel.	`[ac1, pp1]`
`titleVariances`	Abweichungen, die im Titel angezeigt werden.	`"dac1_pp1_pos"`
`titleDateFormat`	Datumsformat. Werte: `year`, `quarter`, `month`, `week`, `day`, `time`, `auto`, `none`.	`quarter`
`titleDateLink`	Datumsverknüpfung. Werte: `avg`, `interval`, `cum`, `start`, `end`, `ytd`, `ytg`, `mat`, `none`.	`end`
`titleOrder`	Sortierung der Titel-Elemente.	`category`
`titleOrderDirection`	Sortierrichtung: `asc` oder `desc`.	`asc`
`i18nNamespace`	Namespace für die Internationalisierung; vererbt sich an alle Kinder der Seite/Card (Kapitel 208).	`_system`

03.02 Beispiel: Vollständig konfigurierter Seitenkopf

```
spec:
  titleBusinessUnit: "BinoBI – Vertrieb"
  titleDateStart: 2026-01-01
  titleDateEnd: 2026-09-30
  titleMeasures:
    - name: "Bruttoumsatz"
      unit: "mEUR"
    - name: "Performance"
      unit: "%"
  titleScenarios: [ac1, fc1, pp1]
  titleDateFormat: quarter
  titleDateLink: ytd
  titleOrder: category
  titleOrderDirection: asc
```

> **Tipp:** Wenn du LayoutCards mit eigenen Titeln verwendest, achte darauf, Kennzahlen und Szenarien möglichst weit „auszuklammern“, also so weit wie möglich in den Seitentitel zu verschieben. Wenn beide LayoutCards auf der Seite beispielsweise AC und Bruttoumsatz enthalten, fasse diese Informationen im Seitentitel zusammen und wiederhole sie nicht noch einmal in den Kartentiteln.

03.03 `titleDateLink` – Start- und Enddatum verknüpfen

Die beiden Datumsfelder werden leicht verwechselt:

- `titleDateFormat` legt fest, **wie ein einzelnes Datum geschrieben** wird (`quarter` → `3Q 2026`).
- `titleDateLink` legt fest, **wie Start- und Enddatum zu einer IBCS-Notation verbunden** werden.

`[Datum]` steht dabei für das formatierte `titleDateStart` , `[Datum2]` für das formatierte `titleDateEnd` .

Wert	Notation	Bedeutung
`none`	`[Datum]`	Einzelnes Datum ohne Verknüpfung (Standard).
`start`	`.[Datum]`	Der Zeitraum beginnt mit diesem Datum.
`end`	`[Datum].`	Der Zeitraum endet mit diesem Datum (Stichtag).
`interval`	`[Datum]..[Datum2]`	Zeitspanne von … bis.
`cum`	`[Datum]_[Datum2]`	Kumuliert über die Zeitspanne.
`ytd`	`_[Datum]`	Year to date – kumuliert bis zum Datum.
`ytg`	`[Datum]_`	Year to go – ab dem Datum bis zum Periodenende.
`mat`	`~[Datum]`	Moving annual total – gleitende Jahressumme.
`avg`	`Ø[Datum]`	Durchschnitt über den Zeitraum.

Bleibt das Feld leer, entfällt die Datumszeile im Titel ganz.

Beispiel: `titleDateFormat: quarter` , `titleDateStart: 2026-01-01` , `titleDateEnd: 2026-09-30` und `titleDateLink: interval` ergeben `1Q..3Q 2026` – die Jahreszahl steht nur einmal am Ende, wenn Start und Ende im selben Jahr liegen.

---

04 – Der Nachrichtenbereich (Action Title)

Über den `children`-Bereich wird optional ein **Nachrichtenbereich** angezeigt. Er eignet sich für einleitende Erklärungen, Kontextinformationen oder Hinweise.

Feld	Beschreibung
`messageText`	Text, der im Nachrichtenbereich angezeigt wird. Unterstützt **Markdown**-Formatierung (Fettdruck, Kursiv, Listen etc.). Eine Übersicht der unterstützten Markdown-Elemente findest du in Kapitel 206, Abschnitt 03.
`messageImage`	Referenz auf ein Asset-Bild (z. B. `asset:logo`), das neben dem Text eingeblendet wird.

04.01 Beispiel

```
spec:
  messageImage: asset:logo
  messageText: >-
    Dieser Bericht zeigt die **Umsatzentwicklung** nach Region
    für das dritte Quartal 2026.
  children:
    - kind: Table
      spec:
        dataset: bb_example_dataset
```

> **Hinweis:** Asset-Referenzen folgen dem Format `asset:<name>`, wobei `<name>` dem `metadata.name` des Asset-Manifests entspricht (z. B. `logo` → `resources/assets/logos/logo.yaml`).

---

05 – Die Fußzeile

Jede Seite kann eine optionale Fußzeile erhalten. Sie wird am unteren Seitenrand angezeigt und eignet sich für Seitenzahlen, Vertraulichkeitshinweise oder Berichtsnamen.

Feld	Beschreibung	Beispiel
`footerText`	Text in der Fußzeile.	`"Nur für den internen Gebrauch"`

Feld	Beschreibung	Beispiel
`footerDisplayPageNumber`	Seitenzahl anzeigen (`true` / `false`).	`true`
`pageNumber`	Seitenzahl manuell überschreiben.	`42`

Beispiel

```
spec:
  footerDisplayPageNumber: true
  footerText: "BinoBI Quartalsbericht – Vertraulich"
```

---

06 – Das Seitenlayout (`pageLayout`)/ Das Cardlayout (`cardLayout`)

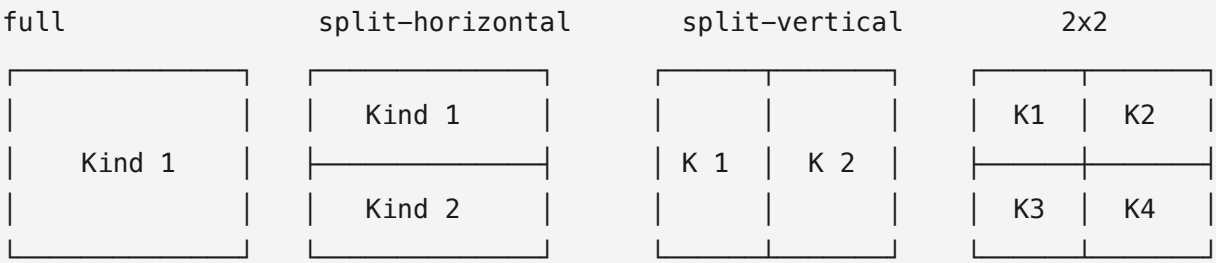
Sowohl bei der LayoutCard als auch bei der LayoutPage bestimmst du das Layout über die Felder `pageLayout` bzw. `cardLayout` . Mit diesen Attributen legst du fest, wie der Inhaltsbereich aufgeteilt wird. So steuerst du, ob die Kind-Komponenten übereinander, nebeneinander oder in einem Raster angeordnet werden.

06.01 Verfügbare Layouts

Wert	Aufteilung	Beschreibung
`full`	1 Bereich	Der gesamte Inhaltsbereich steht einer einzelnen Komponente zur Verfügung. Das ist der Standard, wenn `pageLayout` nicht angegeben wird.
`split-horizontal`	2 Bereiche übereinander	Oben und unten – ideal für eine Tabelle über einem Chart.
`split-vertical`	2 Bereiche nebeneinander	Links und rechts – ideal für zwei Tabellen oder eine Tabelle neben einem Chart.
`1-over-2`	Oben 1, unten 2	Eine Komponente oben, darunter zwei nebeneinander.
`1-over-3`	Oben 1, unten 3	Eine Komponente oben, darunter drei nebeneinander.
`2-over-1`	Oben 2, unten 1	Zwei Komponenten oben nebeneinander, darunter eine Komponente.

Wert	Aufteilung	Beschreibung
`2x2`	4 Bereiche im Raster	Zwei Zeilen mit je zwei Spalten – ideal für Dashboard-Seiten mit vier Kennzahlenblöcken.
`3-over-1`	Oben 3, unten 1	Drei Komponenten oben nebeneinander, darunter eine Komponente.
`3x3`	9 Bereiche im Raster	Drei Zeilen mit je drei Spalten – für komplexere Dashboards.
`4x4`	16 Bereiche im Raster	Vier Zeilen mit je vier Spalten – für sehr große oder detaillierte Layouts.
`custom-template`	CSS grid-template-areas	Im Custom-Template-Modus kannst du ein frei definiertes Raster auf Basis von CSS grid-template-areas erstellen. Das gewünschte Raster wird als String im Feld pageCustomTemplate/ cardCustomTemplate hinterlegt.

06.02 Visueller Überblick



06.03 cardCustomTemplate

Das Attribut `cardCustomTemplate` bzw. `pageCustomTemplate` verwendet die

CSS `grid-template-areas` Raster (Grid). In einem String werden Zeilenweise die einzelnen Zellen benannt. Benachbarte Zellen mit dem gleichen Namen werden als eine Zelle interpretiert.

Beispiel:

```
pageCustomTemplate: |
  "A A"
  "B D"
  "C D"
```

In diesem Beispiel gibt es ein Grid mit zwei Spalten und zwei Zeilen. Die erste Zeile ist für den Header reserviert (über beide Spalten), die zweite Zeile teilt sich in Sidebar und Main-Bereich auf.

Weitere Informationen und viele Beispiele findest du in der offiziellen W3C-Referenz zu grid-template-areas (https://www.w3.org/TR/css-grid-1/#propdef-grid-template-areas).

06.04 Beispiel: `split-vertical` – zwei Spalten

```
spec:
  pageLayout: split-vertical
  children:
    - kind: Table
      spec:
        dataset: bb_example_dataset
        sumTitle: Umsatz nach Region
        scenarios: [ac1, pp1]
        variances: [dac1_pp1_pos]
        type: sum
    - kind: ChartStructure
      spec:
        dataset: bb_example_dataset
        chartTitle: Top 10 Regionen
        scenarios: [ac1, pp1]
        variances: [dac1_pp1_pos]
```

Die erste Komponente (Table) erscheint links, die zweite (ChartStructure) rechts.

06.05 Beispiel: `2x2` – vier Quadranten

```
spec:
  pageLayout: 2x2
  pageFormat: xga
  pageOrientation: landscape
  children:
    - kind: Table
      spec:
        dataset: bb_example_dataset
    - kind: ChartStructure
      spec:
        dataset: bb_example_dataset
```

```
        chartTitle: Quadrant oben rechts
    - kind: ChartTime
      spec:
        dataset: bb_example_dataset
        chartTitle: Quadrant unten links
    - kind: Table
      spec:
        dataset: bb_example_dataset
```

Die Kinder werden in Leserichtung zugeordnet: oben-links, oben-rechts, unten-links, unten-rechts.

06.06 Beispiel: `custom-template`

```
spec:
  pageLayout: custom-template
  pageCustomTemplate: |
    "A B C"
    "A D D"
  pageFormat: xga
  pageOrientation: landscape
  children:
    # 1. Kind -> A (linke Spalte über beide Zeilen)
    - kind: Table
      spec:
        dataset: bb_example_dataset
        type: sum
    # 2. Kind -> B (oben Mitte)
    - kind: ChartStructure
      spec:
        dataset: bb_example_dataset
        chartTitle: Bereich B
    # 3. Kind -> C (oben rechts)
    - kind: ChartTime
      spec:
        dataset: bb_example_dataset
        chartTitle: Bereich C
    # 4. Kind -> D (untere Zeile über Mitte und rechts)
    - kind: Table
      spec:
```

```
        dataset: bb_example_dataset
        type: sum
```

Die Kinder werden den Zellen in der Reihenfolge zugeordnet, in der die Zellnamen im Template zum ersten Mal auftauchen: erstes Kind → `A`, zweites → `B`, drittes → `C`, viertes → `D`.

> **Zwei Stolpersteine:** Der Wert heißt `custom-template` mit Bindestrich – `customTemplate` weist das Schema zurück. Und `pageCustomTemplate` ist ein einzelner String: schreibe die Zeilen als YAML-Block (`|`), sonst ist das YAML ungültig.

> **Wichtig:** Die Anzahl der `children` sollte zur Anzahl der verfügbaren Bereiche passen. Ein `split-vertical`-Layout erwartet zwei Kinder, ein `2x2`-Layout vier.

---

07 – LayoutCards – Inhaltsblöcke mit eigenem Titel

Eine **LayoutCard** ist ein Container, der eine oder mehrere Komponenten gruppiert und ihnen einen eigenen Titelbalken gibt. Du kannst dir eine LayoutCard als „Bericht im Bericht" vorstellen – mit eigenem Titel, eigenen Szenarien, eigenem Zeitraum und optionalem Logo.

LayoutCards sind besonders nützlich, wenn eine Seite mehrere inhaltlich getrennte Blöcke zeigen soll, die jeweils einen eigenen Kontext haben.

07.01 Grundstruktur

```
children:
  - kind: LayoutCard
    metadata:
      name: karte_umsatz
    spec:
      titleBusinessUnit: "Vertrieb Deutschland"
      titleDateStart: 2026-01-01
      titleDateEnd: 2026-09-30
      titleScenarios: [ac1, pp1]
      titleDateFormat: quarter
      titleDateLink: end
      titleImage: logo
      children:
        - kind: Table
          spec:
```

```
            dataset: bb_example_dataset
            sumTitle: Umsatzübersicht
            scenarios: [ac1, pp1]
            type: sum
```

Layout 1: Layout Überblick

07.02 Verfügbare Felder einer LayoutCard

Feld	Beschreibung
`title`	Freitext-Titel der Card.
`description`	Optionale Beschreibung, die unter dem Titel angezeigt wird.
`titleImage`	Referenz auf ein Asset (z. B. `logo`), das im Titelbereich der Card eingeblendet wird.
`titleBusinessUnit`	Organisationseinheit, die im Card-Titel angezeigt wird.
`titleDateStart`	Startdatum des Berichtszeitraums für diese Card.
`titleDateEnd`	Enddatum des Berichtszeitraums für diese Card.
`titleScenarios`	Szenarien, die im Card-Titel dargestellt werden (z. B. `[ac1, pp1]`).
`titleDateFormat`	Datumsformat im Titel: `year`, `quarter`, `month`, `week`, `day`, `time`, `auto`, `none`.
`titleDateLink`	Datumsverknüpfung: `avg`, `interval`, `cum`, `start`, `end`, `ytd`, `ytg`, `mat`, `none`.
`titleMeasures`	Kennzahlen mit Name und Einheit, die im Titel gezeigt werden.
`titleOrder`	Sortierung der Titel-Elemente.
`titleOrderDirection`	Sortierrichtung: `asc` oder `desc`.
`children`	Liste der Komponenten innerhalb der Card (Table, ChartStructure, ChartTime, Text etc.).

07.03 Beispiel: Zwei LayoutCards nebeneinander

Das folgende Beispiel zeigt eine Seite mit `split-vertical`-Layout, bei der jede Hälfte eine eigene LayoutCard mit unabhängigem Titel und Zeitraum enthält:

```
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: dashboard_zwei_karten
  description: Dashboard mit zwei unabhängigen Karten
spec:
  pageFormat: xga
  pageOrientation: landscape
  pageLayout: split-vertical
  children:
    - kind: LayoutCard
      metadata:
        name: karte_top_shops
      spec:
        titleBusinessUnit: "Top 10 Shops"
        titleDateStart: 2026-07-01
        titleDateEnd: 2026-09-30
        titleScenarios: [ac1, pp1]
        titleDateFormat: quarter
        titleDateLink: end
        titleImage: logo
        children:
          - kind: Table
            spec:
              dataset: bb_example_dataset
              sumTitle: Nach Umsatz
              scenarios: [ac1, pl1]
              variances: [dac1_pp1_pos, drac1_pp1_pos]
              barColumns: [dac1_pp1_pos, drac1_pp1_pos]
              type: sum
              limit: 10
              order: ac1
              orderDirection: desc
    - kind: LayoutCard
      metadata:
        name: karte_alle_shops
      spec:
        titleBusinessUnit: "Alle Shops"
        titleDateStart: 2020-01-01
        titleDateEnd: 2026-10-31
        titleScenarios: [ac1, pl1, fc1]
```

```
        titleDateFormat: quarter
        titleDateLink: interval
        children:
          - kind: ChartTime
            spec:
              dataset: bb_example_dataset
              chartTitle: Umsatzentwicklung
              scenarios: [ac1]
              measureScale: k
              measureUnit: EUR
              dateInterval: quarter
```

Auf diese Weise kann die linke Karte ein Quartal im Detail zeigen, während die rechte Karte den gesamten Zeitverlauf über mehrere Jahre darstellt – jeweils mit eigenem Titelbereich.

07.04 LayoutCard vs. direkte Komponenten

Du kannst Komponenten direkt als `children` der LayoutPage einhängen oder sie in LayoutCards verpacken. Wann welcher Ansatz sinnvoll ist:

Ansatz	Wann verwenden
Direkte Komponenten	Einfache Seiten mit einer einzelnen Visualisierung oder wenn alle Komponenten denselben Kontext (Zeitraum, Szenarien) teilen.
LayoutCards	Wenn die Seite mehrere inhaltlich getrennte Blöcke zeigen soll, die jeweils eigene Titel, Zeiträume oder Szenarien haben.

---

08 – Verfügbare Kind-Komponenten

Im `children`-Array einer LayoutPage oder LayoutCard kannst du verschiedene Komponenten-Typen verwenden. Hier eine Übersicht der wichtigsten:

`kind`	Beschreibung	Typischer Einsatz
`Table`	Datentabelle mit Szenarien, Abweichungen und Inline-Balken.	Detailtabellen, Kennzahlenübersichten
`ChartStructure`	Strukturdiagramm (horizontale Balken) für Rangfolgen und Vergleiche.	Top-10-Listen, Portfoliovergleiche

`kind`	Beschreibung	Typischer Einsatz
`ChartTime`	Zeitreihendiagramm für Entwicklungen über Zeit.	Umsatzentwicklung, Trendanalysen
`Text`	Text-Komponente mit Markdown-Unterstützung. Kann Daten aus DataSets einbetten.	Einleitungen, KPI-Zusammenfassungen
`LayoutCard`	Container mit eigenem Titelbereich, der weitere Komponenten gruppiert.	Dashboard-Kacheln
`Grid`	Raster-Layout mit benannten Zeilen und Spalten.	Matrixdarstellungen, Vergleichsübersichten
`Tree`	Baumdiagramm mit Knoten und Kanten für hierarchische Zusammenhänge.	Treiberanalysen, Organigramme
`Label`	Einfacher HTML-Block für freie Gestaltung.	KPI-Boxen, Beschriftungen

> **Hinweis:** `Table` und `ChartStructure` wurden in Kapitel 101 bereits eingeführt. Die Komponenten `ChartTime`, `Grid` und `Tree` werden in späteren Kapiteln im Detail behandelt.

---

09 – Parameter und dynamische Seiten

LayoutPages können **Parameter** definieren, die beim Einbinden in ein ReportArtefact übergeben werden. So kannst du dieselbe Seite mit unterschiedlichen Daten rendern – zum Beispiel für verschiedene Regionen oder Jahre.

09.01 Parameter deklarieren

```
metadata:
  name: seite_mit_params
  params:
    - name: YEAR
      description: Berichtsjahr
      type: date
      required: true
    - name: REGION
      description: Standort
```

```
      type: select
      options:
        items:
          - label: Frankfurt
            value: frankfurt
          - label: Hamburg
            value: hamburg
      required: true
```

09.02 Parametertypen

Typ	Beschreibung
`date`	Ein Datumswert (z. B. `2026-12-31`).
`select`	Auswahl aus einer vordefinierten Liste (`options.items`).

09.03 Parameter verwenden

Parameter werden mit `${PARAMETERNAME}` in Textfeldern referenziert:

```
spec:
  titleBusinessUnit: "Umsatzbericht ${REGION}"
```

09.04 Parameter beim Einbinden übergeben

Im ReportArtefact gibst du die konkreten Werte für jede Seite mit:

```
layoutPages:
  - page: seite_mit_params
    params:
      YEAR: 2026-12-31
      REGION: frankfurt
  - page: seite_mit_params
    params:
      YEAR: 2026-12-31
      REGION: hamburg
```

So entsteht ein PDF mit zwei Seiten – eine für Frankfurt und eine für Hamburg – obwohl nur ein einziges LayoutPage-Manifest existiert.

> **Tipp:** Für `select`-Parameter steht zusätzlich die Variable `${PARAMETERNAME_LABEL}` mit dem benutzerfreundlichen Anzeigenamen zur Verfügung.

---

10 – Alles zusammen: ein vollständiges Dashboard

Das folgende Beispiel zeigt eine komplette Dashboard-Seite im Querformat mit Seitenkopf, Nachrichtenbereich, zwei LayoutCards und Fußzeile:

```
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: bn102_page_and_card_layout
  description: Vollständiges Dashboard-Beispiel
spec:
  pageFormat: a4
  pageOrientation: landscape
  pageLayout: split-horizontal

  # Seitenkopf
  titleBusinessUnit: "BinoBI – Example Department"
  titleDateStart: 2026-07-01
  titleDateEnd: 2026-09-30
  titleMeasures:
    - name: "Revenue"
      unit: "kEUR"
  titleScenarios: [ac1, pp1]
  titleDateFormat: quarter
  titleDateLink: end
  titleOrder: category
  titleOrderDirection: asc

  # Nachrichtenbereich
  messageImage: asset:logo
  messageText: >-
    Quartalsbericht Q3/2026 **Umsatzentwicklung** nach Region
    mit Vergleich zum Vorjahreszeitraum.

  # Fußzeile
  footerDisplayPageNumber: true
```

```
footerText: "Vertraulich – nur für den internen Gebrauch"

# Inhaltsbereich
children:
  - kind: LayoutCard
    metadata:
      name: karte_tabelle
    spec:
      titleBusinessUnit: "Top 10 Standorte"
      titleDateStart: 2026-07-01
      titleDateEnd: 2026-09-30
      titleScenarios: [ac1, pp1]
      titleDateFormat: quarter
      titleDateLink: end
      children:
        - kind: Table
          spec:
            dataset: bb_example_dataset
            sumTitle: Nach Umsatz
            level: category
            order: ac1
            orderDirection: desc
            measureScale: k
            measureType: currency
            measureUnit: EUR
            limit: 10
            scenarios: [ac1, pl1]
            variances: [dac1_pp1_pos, drac1_pp1_pos]
            barColumns: [dac1_pp1_pos, drac1_pp1_pos]
            type: sum
  - kind: LayoutCard
    metadata:
      name: karte_chart
    spec:
      titleBusinessUnit: "Kumulierter Umsatz nach Region"
      titleDateStart: 2020-01-01
      titleDateEnd: 2026-09-30
      titleScenarios: [ac1]
      titleDateFormat: quarter
      titleDateLink: cum
      children:
```

```
      - kind: ChartStructure
        spec:
          dataset: bb_example_dataset
          level: category
          order: ac1
          orderDirection: desc
          measureScale: k
          measureUnit: EUR
          limit: 5
          scenarios: [ac1, pp1]
          variances: [dac1_pp1_pos]
```

---

11 Zusammenfassung

In diesem Kapitel hast du gelernt, wie du Berichtsseiten gestaltest:

1. **Seitenformat und Ausrichtung** – Mit `pageFormat` und `pageOrientation` das Grundformat festlegen.
2. **Seitenlayout** – Mit `pageLayout` den Inhaltsbereich aufteilen (`full` , `split-horizontal` , `split-vertical` , `2x2`).
3. **LayoutCards** – Eigenständige Inhaltsblöcke mit eigenem Titelbereich, die Komponenten gruppieren.
4. **Seitenkopf** – Organisationseinheit, Zeitraum, Szenarien und Kennzahlen im Seitentitel.
5. **Nachrichtenbereich** – Optionaler Bereich mit Text (Markdown) und Bild für Erläuterungen.
6. **Fußzeile** – Seitenzahlen und Freitext am unteren Seitenrand.
7. **Kind-Komponenten** – Übersicht der verfügbaren Visualisierungstypen (Table, ChartStructure, ChartTime, Text, Grid, Tree).
8. **Parameter** – Dynamische Seiten, die beim Einbinden mit unterschiedlichen Werten gerendert werden können.

> **Jetzt bist du dran!** Experimentiere mit verschiedenen Layouts: Erstelle eine Seite mit `2x2` -Layout und vier unterschiedlichen Komponenten. Tausche `split-vertical` gegen `split-horizontal` . Verpacke Inhalte in LayoutCards mit verschiedenen Zeiträumen. Starte `bino preview` und beobachte, wie sich jede Änderung sofort im Browser auswirkt. Je mehr du ausprobierst, desto schneller bekommst du ein Gefühl dafür, welche Anordnung für deinen Bericht am besten funktioniert.

105 - Constraints: Manifeste bedingt einbinden

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).

In den bisherigen Kapiteln hast du `metadata.constraints` bereits in einzelnen Beispielen gesehen – etwa um DataSources nur in der Vorschau zu laden oder DataSets an bestimmte Ausgabeformate zu binden. Dieses Kapitel erklärt das Constraint-System von BinoBI im Detail.

Du lernst:

- was Constraints sind und wie sie funktionieren,
- welche Constraint-Ausdrücke zur Verfügung stehen,
- wie du Manifeste an Modi (`preview` , `build` , `serve`) bindest,
- wie du Manifeste an Ausgabeformate (`a4` , `xga` , `hd` etc.) bindest,
- wie du mit Constraints Sampling, Debug-Ansichten und Testdaten steuerst,
- und wie Constraints mit mehreren Bedingungen kombiniert werden.

---

01 Was sind Constraints?

Constraints sind Bedingungen im `metadata` -Block eines Manifests. Sie steuern, **ob** ein Manifest-Dokument in einem bestimmten Kontext eingebunden wird oder nicht. Ein Manifest mit Constraints wird nur dann verarbeitet, wenn **alle** angegebenen Bedingungen erfüllt sind.

```
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: debug_metrics
  constraints:
    - mode==preview
spec:
  query: |
    SELECT COUNT(*) AS zeilen FROM raw_data
  dependencies:
    - raw_data
```

In diesem Beispiel wird das DataSet `debug_metrics` nur eingebunden, wenn BinoBI im `preview`-Modus läuft. Bei `bino build` oder `bino serve` wird es vollständig ignoriert.

01.01 Constraints in einer Table-Komponente

Constraints können in **jedem** Manifest-Typ verwendet werden, der einen `metadata`-Block besitzt – also auch in Komponenten wie `Table`, `ChartStructure` oder `ChartTime`. Um Constraints zu nutzen, muss das Feld `constraints` als Liste im `metadata`-Block des Manifests angegeben werden. BinoBI wertet diese Bedingungen zur Build-Zeit aus und entscheidet, ob die Komponente eingebunden wird oder nicht.

Im folgenden Beispiel werden zwei Tabellen mit demselben Namen definiert. Die erste Variante zeigt eine kompakte Übersicht für die Bildschirm-Vorschau, die zweite eine detaillierte Ansicht für den Druck im A4-Format:

```
# Kompakte Tabelle für Bildschirmformate
---
apiVersion: bino.bi/v1alpha1
kind: Table
metadata:
  name: umsatz_tabelle
  description: "Kompakte Übersicht für Bildschirmdarstellung."
  constraints:
    - spec.format==xga
spec:
  dataset: umsatz_zusammenfassung


# Detaillierte Tabelle für Druckformat A4
---
apiVersion: bino.bi/v1alpha1
kind: Table
metadata:
  name: umsatz_tabelle
  description: "Detaillierte Tabelle für den A4-Druck."
  constraints:
    - spec.format==a4
spec:
  dataset: umsatz_zusammenfassung
```

Beide Tabellen tragen denselben `metadata.name`. Durch die sich gegenseitig ausschließenden Constraints wird je nach Ausgabeformat nur eine der beiden Varianten eingebunden. Entscheidend ist, dass die `constraints`-Liste **innerhalb** des `metadata`-Blocks steht – nur dort wird sie von BinoBI erkannt und ausgewertet.

> **Wichtig:** Constraints werden zur Build-Zeit ausgewertet, nicht zur Laufzeit. Das bedeutet: Ein Manifest mit einem nicht erfüllten Constraint existiert für BinoBI schlicht nicht – es wird weder geladen noch validiert.

---

02 Syntax und Ausdrücke

Constraints werden als Array von Strings im Feld `metadata.constraints` angegeben. Jeder String ist ein Ausdruck, der zu `true` oder `false` ausgewertet wird.

02.01 Unterstützte Operatoren

Operator	Syntax	Beschreibung	Beispiel
`==`	`feld==wert`	Gleichheit: Feld muss exakt dem Wert entsprechen.	`mode==preview`
`!=`	`feld!=wert`	Ungleichheit: Feld darf nicht dem Wert entsprechen.	`mode!=build`
`in`	`feld in [wert1,wert2,...]`	Enthaltensein: Feld muss einem der Werte entsprechen.	`spec.format in [a4,letter]`
`not-in`	`feld not-in [wert1,wert2,...]`	Ausschluss: Feld darf keinem der Werte entsprechen.	`labels.env not-in [dev,staging]`

02.02 Verfügbare Felder

Feld	Beschreibung	Mögliche Werte
`mode`	Der aktuelle Ausführungsmodus von BinoBI.	`preview`, `build`, `serve`
`artefactKind`	Die Art des Artefakts, das gerade verarbeitet wird.	`report`, `screenshot`, `document`, `livereport`

Feld	Beschreibung	Mögliche Werte
`labels.<key>`	Ein Label aus `metadata.labels` des Ziel-Artefakts.	frei definierbar, z. B. `labels.env==prod`
`spec.<feld>`	Ein Feld aus der `spec` des Ziel-Artefakts, am häufigsten `spec.format`.	z. B. `a4`, `letter`, `xga`, `hd`, `fullhd`

Über `labels.<key>` steuern die frei definierbaren Labels des Ziel-Artefakts (Kapitel 102 nutzt `metadata.labels` bereits für den Present Mode), welche Manifest-Varianten geladen werden – z. B. eine kompakte Seiten-Variante nur für Artefakte mit `labels.variant: simple`.

02.03 Strukturierte Schreibweise

Statt des String-Ausdrucks kann jedes Constraint auch als Objekt mit `field` / `operator` / `value` geschrieben werden – nützlich, wenn Werte Sonderzeichen enthalten oder Manifeste maschinell erzeugt werden:

```
metadata:
  constraints:
    - field: labels.env
      operator: "=="
      value: prod
```

02.04 Mehrere Bedingungen (UND-Verknüpfung)

Wenn du mehrere Constraints angibst, müssen **alle** gleichzeitig erfüllt sein (logisches UND):

```
metadata:
  constraints:
    - mode==build
    - spec.format==a4
```

Dieses Manifest wird nur eingebunden, wenn **sowohl** der Modus `build` ist **als auch** das Format `a4`.

---

03 Modus-Constraints: preview, build, serve

Der häufigste Anwendungsfall für Constraints ist die Unterscheidung nach dem aktuellen mode des Artefaktes.

03.01 Nur in der Vorschau (`mode==preview`)

Mit `mode==preview` definierst du Manifeste, die **nur in der Vorschau** aktiv sind – und im produktiven Build ignoriert werden. Typische Anwendungsfälle:

- **Debug-Ansichten** – zusätzliche Tabellen oder Seiten, die nur der Entwicklung dienen.
- **Testdaten** – Inline-DataSources mit Beispieldaten, die den echten Datenbankzugriff ersetzen.
- **Diagnose-DataSets** – Abfragen, die Datenqualität oder -umfang prüfen.

```
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: debug_metrics
  constraints:
    - mode==preview
spec:
  query: |
    SELECT COUNT(*) AS zeilen, MIN(date) AS von, MAX(date) AS bis
    FROM umsatz_csv
  dependencies:
    - umsatz_csv
```

Dieses DataSet erscheint nur in der Vorschau und wird beim `bino build` vollständig ignoriert.

03.02 Nur im Build (`mode==build`)

Manifeste, die ausschließlich für den produktiven PDF-Export gelten:

```
apiVersion: bino.bi/v1alpha1
kind: DataSource
metadata:
  name: fact_sales
  constraints:
    - mode==build
spec:
  type: parquet
  path: ./warehouse/fact_sales/*.parquet
```

03.03 Alles außer Build (`mode!=build`)

Mit `!=` schließt du einen bestimmten Modus aus. `mode!=build` bedeutet: aktiv in `preview` **und** `serve`, aber nicht in `build`:

```
apiVersion: bino.bi/v1alpha1
kind: DataSource
metadata:
  name: fact_sales
  constraints:
    - mode!=build
spec:
  type: parquet
  path: ./warehouse/fact_sales/*.parquet
  sample: "5%"
```

---

04 Sampling mit Constraints steuern

Ein besonders häufiger Anwendungsfall: Während der Entwicklung nur ein Sample der Daten laden, im produktiven Build aber den vollständigen Datensatz verwenden. Dazu definierst du **zwei DataSources mit demselben** `metadata.name`, die sich durch ihre Constraints unterscheiden:

```
# Vorschau: gesampelte Daten für schnelles Iterieren
---
apiVersion: bino.bi/v1alpha1
kind: DataSource
metadata:
  name: fact_sales
  constraints:
    - mode!=build
spec:
  type: parquet
  path: ./warehouse/fact_sales/*.parquet
  sample: "5%"


# Build: vollständiger Datensatz
---
apiVersion: bino.bi/v1alpha1
kind: DataSource
```

```
metadata:
  name: fact_sales
  constraints:
    - mode==build
spec:
  type: parquet
  path: ./warehouse/fact_sales/*.parquet
```

BinoBI wählt anhand des aktuellen Modus die passende Variante aus. Für nachgelagerte DataSets und Komponenten ist der Unterschied transparent – sie referenzieren einfach `fact_sales`.

Mit Hilfe dieses Musters kann die Entwicklung beschleunigt werden bzw. Datenschutzanforderungen können erfüllt werden, indem in der DEV-Umgebung lediglich ein kleines Sample oder sogar ausschließlich speziell vorbereitete Testdaten verwendet werden.

---

05 Format-Constraints: Ausgabeformate unterscheiden

Mit `spec.format` bindest du Manifeste an bestimmte Ausgabeformate. Das ist nützlich, wenn derselbe Bericht als A4-PDF und als Bildschirm-Dashboard exportiert wird und die Daten unterschiedlich aufbereitet werden sollen.

05.01 Beispiel: Detailliert für Druck, kompakt für Bildschirm

```
# Detailliert für Druck (A4/Letter)
---
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: umsatz_zusammenfassung
  constraints:
    - spec.format in [a4,letter]
spec:
  query: |
    SELECT region, product, month, revenue, costs, margin
    FROM umsatz_csv
    GROUP BY region, product, month
  dependencies:
    - umsatz_csv
```

```
# Zusammengefasst für Bildschirm (XGA)
---
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: umsatz_zusammenfassung
  constraints:
    - spec.format==xga
spec:
  query: |
    SELECT region AS category, SUM(revenue) AS ac1
    FROM umsatz_csv
    GROUP BY region
  dependencies:
    - umsatz_csv
```

Beide DataSets tragen denselben Namen `umsatz_zusammenfassung`. Welches aktiv ist, hängt vom Ausgabeformat des aktuellen Artefakts ab.

---

06 Welche Manifest-Typen unterstützen Constraints?

Constraints stehen in allen Manifest-Typen zur Verfügung, die einen `metadata`-Block besitzen:

Manifest-Typ	Typischer Einsatz
`DataSource`	Sampling in der Vorschau, unterschiedliche Datenquellen pro Umgebung.
`DataSet`	Debug-Abfragen nur in der Vorschau, format-spezifische Aggregationen.
`LayoutPage`	Seiten, die nur in bestimmten Formaten oder Modi erscheinen.
`Table`	Tabellen mit unterschiedlicher Detailtiefe je nach Format.
`ChartStructure`	Charts, die nur im Bildschirmformat angezeigt werden.
`ChartTime`	Zeitreihen-Charts mit formatabhängiger Konfiguration.
`ReportArtefact`	Artefakte, die nur für bestimmte Modi gebaut werden.
`Asset`	Assets, die nur in bestimmten Formaten benötigt werden (z. B. hochauflösend).

Manifest-Typ	Typischer Einsatz
`ComponentStyle`	Stile, die nur für bestimmte Formate gelten (z. B. Print vs. Screen).

---

07 Best Practices

07.01 Gleicher Name, unterschiedliche Constraints (Scoped Names)

Das Muster „zwei Manifeste mit demselben `metadata.name` und sich gegenseitig ausschließenden Constraints" ist der empfohlene Weg, um kontextabhängige Varianten zu definieren. BinoBI stellt sicher, dass in jedem Kontext genau eines der Manifeste aktiv ist.

Die Referenz nennt dieses Konzept **Scoped Names**: Die Eindeutigkeit von `metadata.name` gilt **pro Artefakt nach der Constraint-Filterung**, nicht global. Derselbe Name darf beliebig oft vorkommen, solange die Constraints verhindern, dass zwei Dokumente gleichzeitig auf dasselbe Artefakt zutreffen:

```
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: mainPage
  constraints:
    - labels.variant==simple
---
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: mainPage
  constraints:
    - labels.variant==detailed
```

Zwei `ReportArtefact`s mit unterschiedlichem `labels.variant` binden hier unter demselben Seitennamen `mainPage` jeweils eine andere Seiten-Variante ein.

> **Achtung:** Die Bedingungen müssen sich gegenseitig ausschließen (z. B. `mode==build` und `mode!=build`). Überlappen sie, wird das **nicht** gemeldet: `bino lint` bleibt still und `bino build` läuft durch — welche der gleichnamigen Varianten gewinnt, entscheidet dann die Ladereihenfolge, also faktisch der Dateiname. Für DataSource und DataSet bricht der Build wenigstens mit `duplicate ... name` ab, für `LayoutPage` nicht (siehe #47).

07.02 Constraints sparsam einsetzen

Verwende Constraints gezielt. Ein Projekt mit zu vielen bedingten Manifesten wird schnell unübersichtlich. Typische und bewährte Anwendungsfälle:

- **Sampling** für schnellere Vorschau (siehe Abschnitt 04).
- **Debug-DataSets** nur in der Vorschau (siehe Abschnitt 03.01).
- **Format-spezifische Aufbereitung** für Print vs. Screen (siehe Abschnitt 05).

07.03 Constraints dokumentieren

Ergänze Manifeste mit Constraints um eine `metadata.description`, die erklärt, warum das Constraint gesetzt ist:

```
metadata:
  name: fact_sales
  description: "Gesampelte Daten für schnelle Iteration in der Vorschau."
  constraints:
    - mode!=build
```

---

08 Zusammenfassung

In diesem Kapitel hast du das Constraint-System von BinoBI kennengelernt:

1. **Constraints** steuern, ob ein Manifest in einem bestimmten Kontext eingebunden wird.
2. **Modus-Constraints** (`mode==preview`, `mode==build`, `mode!=build`) unterscheiden zwischen Entwicklung und Produktion.
3. **Format- und Artefakt-Constraints** (`spec.format==a4`, `artefactKind==document`, `labels.env==prod`) binden Manifeste an Eigenschaften des Ziel-Artefakts.
4. **Vier Operatoren** stehen zur Verfügung: `==`, `!=`, `in`, `not-in` – als String-Ausdruck oder in strukturierter `field`/`operator`/`value`-Schreibweise.
5. **UND-Verknüpfung** – mehrere Constraints in einem Array müssen alle erfüllt sein.
6. **Scoped Names**: `metadata.name` muss nur pro Artefakt (nach Constraint-Filterung) eindeutig sein – gleichnamige Manifeste mit sich gegenseitig ausschließenden Constraints sind das empfohlene Muster für kontextabhängige Varianten.

> **Jetzt bist du dran!** Erstelle zwei DataSources mit demselben Namen – eine mit `constraints: [mode!=build]` und `sample: "5%"`, die andere mit `constraints:`

> `[mode==build]` ohne Sample. Starte `bino preview` und prüfe, ob nur das gesampelte DataSet aktiv ist. Führe anschließend `bino build` aus und vergleiche die Ergebnisse.

106 - Assets und Bilder

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).

In den bisherigen Kapiteln sind dir Bilder bereits informell begegnet – als Logo im Seitenkopf (Kapitel 104) oder als Bild im Baumknoten (Kapitel 205, sobald du dort ankommst). Dieses Kapitel erklärt das `Asset`-Manifest im Detail: wie du Bilder, Schriftarten und Dateien registrierst und wie du sie aus anderen Manifesten heraus referenzierst.

Du lernst:

- was ein `Asset`-Manifest ist und welche Typen es gibt,
- wie du ein Bild aus einer lokalen Datei, einer URL oder inline als Base64 einbindest,
- wie du ein Asset referenzierst – und warum die Syntax dafür nicht überall gleich aussieht,
- und wie ein durchgängiges Beispiel mit Logo, Flagge und Markdown-Bild funktioniert.

---

01 Das Asset-Manifest

Ein `Asset` registriert eine Bild-, Schrift- oder Dateiressource unter einem Namen, damit andere Manifeste sie referenzieren können, ohne den Dateipfad zu wiederholen:

```
apiVersion: bino.bi/v1alpha1
kind: Asset
metadata:
  name: logo
  description: "Firmenlogo"
spec:
  type: image
  mediaType: image/png
  source:
    localPath: image.png
```

Attribut	Typ	Beschreibung
`type`	string	**Pflichtfeld.** `image`, `font` oder `file`.

Attribut	Typ	Beschreibung
`mediaType`	string	MIME-Type der Ressource, z. B. `image/png`, `image/svg+xml`, `font/woff2`.
`source`	object	Genau eine der drei Quellen unten.

01.01 Die drei Quellen

Quelle	Beschreibung
`localPath`	Pfad relativ zur Asset-Manifest-Datei, z. B. `image.png` oder `1x1/de.svg`.
`remoteURL`	Absolute URL, von der die Ressource zur Build-Zeit geladen wird.
`inlineBase64`	Base64-kodierter Inhalt direkt im Manifest – praktisch für sehr kleine Icons.

Genau eine dieser drei Optionen muss gesetzt sein. `localPath` ist die häufigste Wahl, da die Bilddatei so zusammen mit dem Manifest im Projekt versioniert wird.

---

02 Assets referenzieren: zwei Schreibweisen

Das ist die Stelle, an der die meisten Einsteiger stolpern: **nicht jedes Feld referenziert ein Asset auf die gleiche Weise.**

Kontext	Schreibweise	Beispiel
`messageImage` (LayoutPage)	`asset:<name>`	`messageImage: asset:logo`
Markdown-Bild in `Text` (`value`)	`asset:<name>`	`![Logo](asset:logo)`
`titleImage` (LayoutPage/LayoutCard)	blanker Name	`titleImage: logo`
`ref` auf ein Asset in einem Tree-`Image`-Knoten	blanker Name	`ref: flag_de`

> **Warum zwei Schreibweisen?** `messageImage` und Markdown-Bilder akzeptieren generische URI-Strings – das `asset:`-Schema unterscheidet dort eine Asset-Referenz von einem externen Link oder Pfad. `titleImage` und `ref` sind dagegen bereits als reine Asset-Referenz typisiert, sodass das Schema-Prefix überflüssig ist. Wenn ein Bild nicht

> erscheint, ist eine vertauschte Schreibweise die häufigste Ursache – probiere die jeweils andere.

---

03 Lokale Bilddateien

Der Regelfall: Ein Bild liegt im Projekt (typischerweise unter `resources/assets/`) und wird per `localPath` referenziert.

```
apiVersion: bino.bi/v1alpha1
kind: Asset
metadata:
  name: flag_de
  description: "Flagge von Deutschland"
spec:
  type: image
  mediaType: image/svg+xml
  source:
    localPath: 1x1/de.svg
```

SVGs eignen sich besonders für Icons und Flaggen, da sie verlustfrei skalieren. Für Fotos oder Screenshots sind PNG oder JPEG üblich.

---

04 Remote-Bilder und Inline-Base64

04.01 `remoteURL`

```
apiVersion: bino.bi/v1alpha1
kind: Asset
metadata:
  name: cdn_logo
spec:
  type: image
  mediaType: image/png
  source:
    remoteURL: "https://cdn.example.com/brand/logo.png"
```

BinoBI lädt die Ressource zur Build-Zeit. Beachte die CDN-Limits `BNR_CDN_MAX_BYTES` und `BNR_CDN_TIMEOUT_MS` (siehe `bino.toml [preview.env]`), falls das Bild groß ist oder der Server langsam antwortet.

04.02 `inlineBase64`

```
apiVersion: bino.bi/v1alpha1
kind: Asset
metadata:
  name: tiny_icon
spec:
  type: image
  mediaType: image/png
  source:
    inlineBase64: "iVBORw0KGgoAAAANSUhEUgAA..."
```

Nützlich für sehr kleine, projektinterne Icons, wenn eine zusätzliche Datei den Vorteil nicht rechtfertigt. Für alles, was größer als ein paar Kilobyte ist, ist `localPath` die bessere Wahl – Base64 bläht das YAML-Dokument spürbar auf.

---

05 Assets als eigenständige, wiederverwendbare Komponenten

Wie Table, Text oder ChartStructure kann auch ein Asset als eigene Datei definiert und aus mehreren Manifesten heraus wiederverwendet werden. Genau das nutzt das Projekt bereits für Logo und Länderflaggen:

```
# resources/assets/logos/logo.yaml
apiVersion: bino.bi/v1alpha1
kind: Asset
metadata:
  name: logo
  description: "bino_bi logo"
spec:
  type: image
  mediaType: image/png
  source:
    localPath: image.png
```

```
# resources/assets/pictograms/flags/flags.yaml (Auszug)
apiVersion: bino.bi/v1alpha1
kind: Asset
metadata:
  name: flag_de
  description: "Flagge von Deutschland"
spec:
  type: image
  mediaType: image/svg+xml
  source:
    localPath: 1x1/de.svg
```

Beide werden bereits an mehreren Stellen im Projekt referenziert, etwa als `messageImage: asset:logo` in Kapitel 104.

---

06 Beispiel: Alle drei Referenz-Arten in einer Seite

Das folgende Beispiel kombiniert `messageImage`, ein Markdown-Bild in `Text` und `titleImage` einer `LayoutCard`:

```
spec:
  messageImage: asset:logo
  children:
    - kind: Text
      ref: 106_example_text_asset_image
    - kind: LayoutCard
      ref: 106_example_card_titleimage
```

```
# components/106/106_example_text_asset_image.yaml
apiVersion: bino.bi/v1alpha1
kind: Text
metadata:
  name: 106_example_text_asset_image
spec:
  value: |
    ### Bild aus einem Asset-Manifest

    In Markdown eingebettete Asset-Bilder nutzen das `asset:`-Schema:
```

```
    ![Beispielgrafik](asset:example_illustration)
```

```yaml
# components/106/106_example_card_titleimage.yaml
apiVersion: bino.bi/v1alpha1
kind: LayoutCard
metadata:
  name: 106_example_card_titleimage
spec:
  titleImage: flag_de # blanker Name, kein "asset:"-Prefix
  titleBusinessUnit: "Vertrieb Deutschland"
  titleScenarios: [ac1, pp1]
  titleDateFormat: quarter
  titleDateLink: end
  children:
    - kind: Table
      spec:
        dataset: bb_example_dataset
        sumTitle: Umsatzübersicht
        scenarios: [ac1, pp1]
        type: sum
```

LayoutCard mit titleImage (blanker Asset-Name)

---

07 Zusammenfassung

1. Ein **`Asset`-Manifest** registriert Bilder, Schriften oder Dateien unter einem `metadata.name`.
2. Genau eine der drei Quellen `localPath`, `remoteURL` oder `inlineBase64` muss gesetzt sein.
3. **`asset:<name>`** wird in generischen URI-Feldern verwendet (`messageImage`, Markdown-Bilder in `Text`).
4. **Blanke Namen** werden verwendet, wo das Feld bereits als Asset-Referenz typisiert ist (`titleImage`, Tree-`Image`-`ref`).
5. Assets lassen sich wie andere Komponenten als **eigenständige Dateien** definieren und projektweit wiederverwenden.

> **Jetzt bist du dran!** Lege ein neues Asset für ein eigenes Bild an (z. B. ein Icon aus `resources/assets/`), referenziere es einmal über `messageImage` und einmal über ein Markdown-Bild in einer `Text`-Komponente. Prüfe in `bino preview`, ob beide Varianten korrekt rendern.

107 - Grid

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).

In Kapitel 104 hast du `pageLayout` / `cardLayout` mit festen Rastern (`2x2` , `3x3` , …) und mit `custom-template` kennengelernt. Die `Grid` -Komponente löst ein verwandtes, aber anderes Problem: eine **Vergleichsmatrix**, bei der jede Zelle über einen benannten Zeilen- und Spaltenkopf adressiert wird – z. B. “Produktkategorie × Quartal” oder, wie in diesem Kapitel, “Kennzahl × Darstellung”.

Du lernst:

- wann du `Grid` statt `custom-template` verwendest,
- wie `rowHeaders` und `columnHeaders` funktionieren,
- wie Kind-Elemente per `row` / `column` in Zellen platziert werden,
- und wie sich `Grid` von den Layout-Rastern aus Kapitel 104 unterscheidet.

---

01 Minimalkonfiguration

Ein `Grid` braucht mindestens `rowHeaders` , `columnHeaders` und `children` :

```
- kind: Grid
  spec:
    rowHeaders: [electronics, apparel]
    columnHeaders: [q1, q2]
    children:
      - row: electronics
        column: q1
        kind: Table
        spec:
          dataset: bb_example_dataset
```

`rowHeaders` / `columnHeaders` akzeptieren entweder eine einfache Liste von Strings (die gleichzeitig als Label **und** als ID dienen) oder eine Liste von Objekten mit getrenntem `label` und `id` :

```
rowHeaders:
  - label: "Nach Umsatz"
    id: nach_umsatz
  - label: "Kumuliert"
    id: kumuliert
```

---

02 Darstellung des Rasters

Attribut	Typ	Standard	Beschreibung
`chartTitle`	string	–	Titel über dem gesamten Grid.
`showRowHeaders`	bool	`true`	Zeigt die Zeilenköpfe an.
`showColumnHeaders`	bool	`true`	Zeigt die Spaltenköpfe an.
`showBorders`	bool	`true`	Zeigt Trennlinien zwischen den Zellen.
`rowHeaderWidth`	string	`auto`	CSS-Breite der Zeilenkopf-Spalte, z. B. `auto`, `100px`, `20%`.
`cellGap`	string	`0px`	CSS-Abstand zwischen den Zellen.

---

03 Kind-Elemente platzieren

Jedes Element in `children` gehört über `row` und `column` zu genau einer Zelle. Die Werte müssen einer `id` aus `rowHeaders` / `columnHeaders` entsprechen:

```
children:
  - row: nach_umsatz
    column: tabelle
    kind: Table
    spec:
      dataset: bb_example_dataset
      sumTitle: "Top 5 Regionen"
      order: ac1
      orderDirection: desc
```

```
      limit: 5
      scenarios: [ac1, pp1]
```

Erlaubte `kind`-Werte für Grid-Kinder: `Text`, `Table`, `ChartStructure`, `ChartTime`, `Tree`, `Image`. Anders als bei `LayoutPage` / `LayoutCard` gehören **`Grid` und `LayoutCard` selbst nicht** zu den erlaubten Kind-Kinds – ein Grid kann also keine verschachtelten Grids oder Cards enthalten. Wie bei jedem anderen Kind-Element kannst du `ref` (plus optionale `spec`-Overrides) nutzen, um eine Zelle aus einer eigenständigen Komponente zu befüllen (siehe Kapitel 106, Abschnitt 05).

Mehrere Zellen dürfen dieselbe Komponente unterschiedlich konfiguriert zeigen – jede Zelle hat ihre eigene `spec`.

---

04 Grid vs. `custom-template`

Kriterium	**`Grid`**	**`pageLayout: custom-template`**
Adressierung	Über benannte Zeilen-/Spalten-`id`, unabhängig von der Position.	Über Buchstaben in einem `grid-template-areas`-String (Kapitel 104).
Köpfe (Labels)	Eingebaut (`rowHeaders` / `columnHeaders`, optional ausblendbar).	Keine – Beschriftung musst du selbst per `Text` / `titleBusinessUnit` lösen.
Typischer Anwendungsfall	Vergleichsmatrix: dieselbe Struktur über mehrere Dimensionen (Produkt × Quartal, Kennzahl × Ansicht).	Freie Anordnung von Komponenten unterschiedlicher Größe auf einer Seite.
Verschachtelung	Keine `Grid` / `LayoutCard`-Kinder.	`LayoutCard` als Kind möglich.

Als Faustregel: Wenn deine Seite eine **Matrix aus vergleichbaren Zellen** mit Zeilen- und Spaltenbeschriftung ist, greife zu `Grid`. Wenn du unterschiedlich große Komponenten frei auf der Seite anordnen willst, bleibt `custom-template` die richtige Wahl.

---

05 Beispiel: Tabelle und Chart im Vergleich

Das folgende Grid stellt dieselben zwei Sichten (Top-5-Regionen “nach Umsatz” und “kumuliert”) jeweils als Tabelle und als Chart nebeneinander dar:

```
apiVersion: bino.bi/v1alpha1
kind: Grid
metadata:
  name: 107_example_grid
spec:
  chartTitle: "Umsatz nach Region: Tabelle und Chart im Vergleich"
  rowHeaders:
    - label: "Nach Umsatz"
      id: nach_umsatz
    - label: "Kumuliert"
      id: kumuliert
  columnHeaders:
    - label: "Tabelle"
      id: tabelle
    - label: "Chart"
      id: chart
  showRowHeaders: true
  showColumnHeaders: true
  showBorders: true
  rowHeaderWidth: auto
  cellGap: 8px
  children:
    - row: nach_umsatz
      column: tabelle
      kind: Table
      spec:
        dataset: bb_example_dataset
        sumTitle: "Top 5 Regionen"
        order: ac1
        orderDirection: desc
        measureScale: k
        measureUnit: EUR
        limit: 5
        scenarios: [ac1, pp1]
        variances: [dac1_pp1_pos]
        type: sum
    - row: nach_umsatz
      column: chart
```

```
      kind: ChartStructure
      spec:
        dataset: bb_example_dataset
        chartTitle: "Top 5 Regionen"
        level: category
        order: ac1
        orderDirection: desc
        measureScale: k
        measureUnit: EUR
        limit: 5
        scenarios: [ac1, pp1]
        variances: [dac1_pp1_pos]
        unitScaling: bn107_chart_scale # ScalingGroup: beide Chart-Zellen teilen
    # ... zweite Zeile "kumuliert" analog, siehe components/107/107_example_grid
```

Beide Chart-Zellen referenzieren hier über `unitScaling` dieselbe **`ScalingGroup`** (`bn107_chart_scale`, definiert in `styles/bn107_chart_scale.yaml`), statt jeweils `auto` zu verwenden – so ergeben gleiche Werte in beiden Zellen exakt gleich lange Balken. Das Manifest dahinter ist bewusst klein; Kapitel 207, Abschnitt 06 erklärt es im Detail.

Grid: Tabelle und Chart im Vergleich, zwei Zeilen und zwei Spalten

---

06 Zusammenfassung

1. **`Grid`** stellt eine Vergleichsmatrix mit benannten Zeilen- und Spaltenköpfen dar.
2. `rowHeaders` / `columnHeaders` akzeptieren einfache String-Listen oder `{label, id}`-Objekte.
3. Jedes Kind-Element wird über `row` / `column` (passend zu einer Header-`id`) einer Zelle zugeordnet.
4. Erlaubte Kind-Kinds: `Text`, `Table`, `ChartStructure`, `ChartTime`, `Tree`, `Image` – **kein** `Grid` oder `LayoutCard`.
5. Nutze `Grid` für vergleichbare Zellen über zwei Dimensionen, `custom-template` (Kapitel 104) für freie Anordnung.

> **Jetzt bist du dran!** Erstelle ein `Grid` mit drei Spalten (z. B. drei Regionen) und zwei Zeilen (Tabelle und Chart), basierend auf `bb_example_dataset`. Blende anschließend `showBorders` aus und beobachte den Unterschied in `bino preview`.

108 - Bino-Werkzeuge für den Alltag

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).

Die bisherigen Kapitel haben sich auf `bino preview`, `bino lint` und `bino build` konzentriert – die Kommandos, mit denen du täglich arbeitest. Daneben bringt die CLI eine Reihe von Werkzeugen für Installation, Wartung und Projekt-Bootstrapping mit. Dieses Kapitel sammelt sie an einem Ort.

Du lernst:

- wie du den Cache leerst, wenn sich Daten nicht aktualisieren,
- wie du Chrome und die Template-Engine mit `bino setup` installierst und aktualisierst,
- wie du `bino` selbst per `bino update` aktuell hältst,
- wie du Shell-Autovervollständigung einrichtest,
- wie du Projekt- und Versionsinformationen abfragst,
- wie du mit `bino init` aus einer Vorlage (auch einer entfernten) startest,
- und was hinter der Editor- (`bino lsp`) und KI-Agenten-Integration (`bino mcp`) steckt.

---

01 Cache-Verwaltung: `bino cache clean`

BinoBI cached Datensatz-Ergebnisse und andere Zwischenstände in `.bino/cache`, um `bino preview` beim erneuten Laden zu beschleunigen. Wenn sich Daten nicht wie erwartet aktualisieren – etwa nach einer externen Änderung an einer Datenbank, die BinoBI nicht selbst beobachten kann –, hilft ein manuelles Leeren:

```
bino cache clean                # löscht ./.bino/cache im aktuellen Projekt
bino cache clean --work-dir ../anderes-projekt
bino cache clean --global       # löscht zusätzlich den globalen Cache unter ~/.
```

Flag	Beschreibung
`--global`	Löscht zusätzlich den globalen Cache unter `~/.bino/`.
`-w, --work-dir`	Arbeitsverzeichnis, das `.bino/cache` enthält (Default `.`).

> **Hinweis:** `.bino/cache` und `.bncache/` sind Build-Artefakte und gehören nicht ins Git-Repository (siehe `.gitignore`). Ein gelöschter Cache wird beim nächsten `bino preview` / `bino build` automatisch neu aufgebaut – es geht nichts verloren außer etwas Wartezeit.

---

02 Abhängigkeiten installieren: `bino setup`

`bino setup` lädt die beiden externen Komponenten herunter, die BinoBI zum Rendern benötigt: den Chrome-Headless-Shell (für PDF-Export) und die `bn-template-engine`.

```
bino setup                                          # beide Komponenten sicherst
bino setup --template-engine                        # nur die Template-Engine
bino setup --template-engine --engine-version v1.2.3 # eine bestimmte Engine-Ver
bino setup --dry-run                                 # zeigt an, was passieren w
```

`bino.toml` referenziert die geladene Engine-Version im Feld `engine-version` – nach einem `bino setup --template-engine --engine-version ...` solltest du diesen Wert im Projekt entsprechend anpassen, damit Team-Kolleg:innen dieselbe Version verwenden.

> **Die Engine-Version entscheidet, welche Komponenten es gibt.** Die CLI validiert Manifeste gegen ihr eigenes Schema – ob die Template-Engine die jeweilige Komponente auch *rendern* kann, prüft sie dabei nicht. Die Folge: Ein Manifest validiert, der Build läuft durch, und das Diagramm bleibt im PDF trotzdem leer.
>
> Konkret betrifft das die XY-Komponenten `ChartScatter`, `ChartBubble` und `ChartBullet` (Kapitel 211 und 212). Sie brauchen mindestens **`v1.0.0-next.19`**; mit der `v1.0.0-alpha.15` fehlen die zugehörigen Elemente im Engine-Bundle vollständig.
>
> ```
> bino setup --template-engine --engine-version v1.0.0-next.22
> ```
>
> Danach den Wert in `bino.toml` angleichen. Bei einem leeren Diagramm ist die Engine-Version deshalb immer die erste Hypothese – nicht die letzte.

---

03 Bino selbst aktualisieren: `bino update`

```
bino update
```

Aktualisiert sowohl die `bino`-Binary als auch die Template-Engine auf die jeweils neueste veröffentlichte Version von GitHub. Praktisch als `pre-preview`-Hook in `bino.toml`, wenn ein Team stets auf dem neuesten Stand entwickeln soll (siehe Kapitel 210, Abschnitt 02) – in diesem Projekt ist genau das als Beispiel bereits vorbereitet, aber auskommentiert:

```
[preview.hooks]
#pre-preview = ["bino update"]
```

---

04 Shell-Autovervollständigung: `bino completion`

```
bino completion zsh   > ~/.zsh/completions/_bino     # oder bash / fish / powers
```

Erzeugt ein Autovervollständigungs-Skript für die angegebene Shell. Nach dem Einbinden in dein Shell-Profil vervollständigt Tab sowohl Unterkommandos (`bino pre<TAB>` → `preview`) als auch bekannte Flags.

---

05 Projekt- und Versionsinformationen

```
bino version   # bino 0.81.0 (commit f93f64fb, built 2026-07-04T07:24:59Z)
bino about     # Version, Build-Datum und alle direkten Abhängigkeiten mit Lizer
```

`bino about` ist besonders nützlich, um beim Melden eines Fehlers oder beim Prüfen von Lizenzauflagen schnell die exakte Version und alle Bibliotheken zu sehen, auf denen `bino` aufbaut.

---

06 Projekte aus Vorlagen starten: `bino init`

Kapitel 101 hat `bino init -y` bereits kurz gezeigt. `bino init` kann mehr als nur das eingebaute Minimal-Gerüst erzeugen:

```
bino init                              # eingebautes "minimal"-Template, interak
bino init standard                     # eingebautes "standard"-Template (kanoni
bino init owner/repo                   # entferntes GitHub-Template (Standard-Br
bino init owner/repo/subdir#v1.2.0     # Unterordner eines Tags/Branches
```

```
bino init https://example.com/tmpl.zip # Archiv-URL
bino init ./local-template             # lokaler Pfad
```

Flag	Beschreibung
`-d, --directory`	Zielverzeichnis für das neue Projekt (Default `./rainbow-report`).
`--force`	Überschreibt vorhandene Dateien.
`--json`	Gibt das Ergebnis als JSON aus (z. B. für CI-Skripte).
`--language`	Standard-Sprache des Projekts (`en` oder `de`).
`--name` / `--title`	`metadata.name` /Titel für das Beispiel- `ReportArtefact` der eingebauten Vorlagen.
`--set key=value`	Setzt einen Vorlagen-Feldwert (wiederholbar) – nützlich beim nicht-interaktiven Bootstrapping.
`--offline`	Erreicht nie das Netzwerk; erfordert eine bereits gecachte Vorlage.
`--trust`	Überspringt die Sicherheitsabfrage bei einer nicht-kuratierten entfernten Quelle.
`-y, --yes`	Übernimmt alle Defaults und überspringt den interaktiven Wizard.

06.01 Sicherheit bei entfernten Vorlagen

Entfernte Vorlagen (`owner/repo[/subdir]#ref` , Archiv-URLs) werden von GitHub geladen, anhand ihres Commit-SHA gecached – und **führen nie Code aus**. Trotzdem fragt `bino init` bei einer nicht-kuratierten Quelle standardmäßig nach Bestätigung, bevor Dateien auf die Platte geschrieben werden. `--trust` überspringt diese Abfrage (z. B. in einer CI-Pipeline mit bereits geprüfter Quelle), `--offline` verweigert jeden Netzwerkzugriff und bricht ab, wenn die Vorlage nicht bereits im lokalen Cache liegt.

```
bino init acme-corp/report-templates/quarterly#v3 \
  --set title="Q3 Bericht" \
  --set author="Finance Team" \
  --trust -y
```

---

07 Editor- und KI-Integration: `bino lsp` und `bino mcp`

Zwei Befehle, die du selten direkt aufrufst, die aber die Werkzeuge um dich herum antreiben:

07.01 `bino lsp`

```
bino lsp [--work-dir DIR]
```

Startet einen **Language Server** (LSP) über stdio – die Grundlage für Autovervollständigung, Hover-Dokumentation, Diagnosen, Go-to-Definition und Umbenennen in Editoren. Die VS-Code-Extension aus Kapitel 101 nutzt genau diesen Server; für andere LSP-fähige Editoren (Neovim, Helix, …) kannst du ihn manuell einbinden.

07.02 `bino mcp`

```
bino mcp [--work-dir DIR]
```

Startet einen **Model-Context-Protocol-Server** für KI-Agenten (Claude Code, Claude Desktop, Cursor). Der Agent bekommt damit Werkzeuge zum Inspizieren (`list_kinds` , `describe_project` , `get_columns` , `get_rows` , `graph_deps`), Validieren (`validate_project` , `validate_draft`), Autorisieren von Manifesten (`create_manifest` , `edit_manifest` – jede Schreiboperation wird vor dem Speichern gegen das JSON-Schema validiert) und Bauen (`build`). Einrichtung in Claude Code:

```
claude mcp add bino -- bino mcp --work-dir /pfad/zum/report
```

07.03 Der gemeinsame Daemon

Läuft bereits eine `bino` -Instanz für das Projekt (z. B. weil VS Code mit der Extension geöffnet ist), verbinden sich `bino lsp` und `bino mcp` automatisch mit deren Daemon (sichtbar an `.bino/daemon.json`) und teilen sich DuckDB-Session und Datei-Watcher, statt alles doppelt zu laden. `--no-proxy` erzwingt einen eigenständigen Start.

---

08 Zusammenfassung

1. **`bino cache clean [--global]`** leert den Query-Cache, wenn Daten sich nicht aktualisieren.
2. **`bino setup`** installiert/aktualisiert Chrome-Headless-Shell und Template-Engine.
3. **`bino update`** aktualisiert `bino` selbst – praktisch als `pre-preview` -Hook.
4. **`bino completion <shell>`** erzeugt Tab-Vervollständigung für bash/fish/powershell/zsh.

5. **`bino version`** / **`bino about`** zeigen Versions- und Abhängigkeitsinformationen.
6. **`bino init`** kann neben dem eingebauten Minimal-/Standard-Template auch entfernte Vorlagen (GitHub, Archiv-URL, lokaler Pfad) laden – niemals mit Codeausführung, aber mit einer Sicherheitsabfrage bei nicht-kuratierten Quellen.
7. **`bino lsp`** / **`bino mcp`** treiben Editor- (LSP) und KI-Agenten-Integration (MCP) an und teilen sich über den Projekt-Daemon eine DuckDB-Session.

> **Jetzt bist du dran!** Führe `bino cache clean` aus und beobachte in `bino preview --log-sql`, dass Datensatz-Abfragen beim nächsten Laden erneut ausgeführt werden. Erzeuge anschließend mit `bino completion` das Vervollständigungs-Skript für deine Shell und binde es ein.

109 – Eine Seite, viele Berichte: LayoutPage-Parameter

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/guides/layoutpage-params/ (https://cli.bino.bi/guides/layoutpage-params/).

Bis hierhin hat jede `LayoutPage` genau eine Seite ergeben. Sobald derselbe Aufbau aber für mehrere Regionen, Jahre oder Produktlinien gebraucht wird, endet das schnell in Copy-and-paste: fünf fast identische Seiten, die sich nur in einem Filter unterscheiden – und die du bei jeder Änderung fünfmal anfassen musst.

Parameter lösen genau das. Du definierst die Seite einmal mit Platzhaltern und sagst beim Einbinden in das `ReportArtefact`, mit welchen Werten sie gefüllt werden soll.

Du lernst:

- wie du eine `LayoutPage` mit `metadata.params` parametrisierst,
- wie du Platzhalter mit `${NAME}` im Spec verwendest,
- welche Parametertypen es gibt (`string`, `number`, `boolean`, `select`, `date`, `date_time`),
- wie `select`-Parameter über `${NAME_LABEL}` einen lesbaren Anzeigetext liefern,
- wie du dieselbe Seite im `ReportArtefact` mehrfach mit unterschiedlichen Werten einbindest,
- wie Parameter auch auf referenzierten Komponenten funktionieren,
- und warum Parameter in `bino preview` anders aussehen als im Build.

---

01 Das Problem an einem Beispiel

Angenommen, du brauchst denselben Umsatzbericht für Europa und Nordamerika. Ohne Parameter sieht das so aus:

```
- kind: Table
  spec:
    dataset: umsatz
```

```
    filter: "region = 'EU'"
    sumTitle: "Umsatz Europa"
```

… und daneben noch einmal dasselbe mit `US` und `"Umsatz Nordamerika"`. Bei zwei Regionen ist das lästig, bei zwölf Ländern ist es nicht mehr wartbar.

---

02 Die parametrisierte Seite

Parameter werden in **`metadata.params`** deklariert – nicht im `spec`. Das ist wichtig: `params` beschreibt die *Schnittstelle* der Seite, `spec` ihren Inhalt.

```
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: regionsseite
  params:
    - name: REGION
      type: select
      required: true
      description: "Vertriebsregion, die diese Seite zeigt"
      options:
        items:
          - value: "EU"
            label: "Europa"
          - value: "US"
            label: "Nordamerika"
    - name: JAHR
      type: number
      default: "2026"
spec:
  titleBusinessUnit: "${REGION_LABEL} — Umsatz ${JAHR}"
  children:
    - kind: Table
      spec:
        dataset: umsatz
        type: sum
        sumTitle: "Umsatz ${REGION_LABEL}"
        filter: "region = '${REGION}'"
```

```
        scenarios: ["ac1", "pp1"]
        variances: ["dac1_pp1_pos"]
```

Zwei Dinge lohnen einen zweiten Blick:

- **`${REGION}`** liefert den *Wert* (`EU`) – den brauchst du im `filter`, weil er auf die Daten passen muss.
- **`${REGION_LABEL}`** liefert das *Label* (`Europa`) – das gehört in Titel und Überschriften.

Diese Trennung ist der Hauptgrund, `select` statt `string` zu verwenden: technische Schlüssel in den Daten, lesbare Namen in der Ausgabe. Ist zu einem Eintrag kein `label` definiert, gibt `${NAME_LABEL}` einfach den Wert zurück.

---

03 Die Seite mehrfach einbinden

Im `ReportArtefact` wird aus dem schlichten Seitennamen ein Objekt aus `page` und `params`:

```
apiVersion: bino.bi/v1alpha1
kind: ReportArtefact
metadata:
  name: regionsbericht
spec:
  format: xga
  orientation: landscape
  language: de
  filename: regionsbericht.pdf
  title: "Regionsbericht"
  layoutPages:
    - page: regionsseite
      params:
        REGION: "EU"
        JAHR: "2026"
    - page: regionsseite
      params:
        REGION: "US"
        JAHR: "2026"
```

`bino build --artefact regionsbericht` erzeugt daraus ein PDF mit **zwei** Seiten – dieselbe Seitendefinition, zweimal gefüllt. Eine dritte Region ist ab jetzt ein Vierzeiler, keine neue Seite.

> **Hinweis:** Parameterwerte werden als **Strings** übergeben – auch bei `type: number`. Schreibe also `JAHR: "2026"`, nicht `JAHR: 2026`. YAML würde die Zahl sonst als Integer interpretieren, und die Ersetzung erwartet einen String.

► **Zum Ausprobieren:** Vollständiges Beispiel in der Sandbox öffnen (https://sandbox.bino.bi/s/bafesga1erionyq) – enthält Daten, Seite und Artefakt in einer Datei. Über **Fork** erhältst du eine eigene, editierbare Kopie; ändere z. B. `REGION: "US"` in der zweiten Seite und beobachte Titel und Tabelle.

> Quelle im Repo: 109_01_parameter_regionen.yaml

---

04 Die Parametertypen

Typ	Beschreibung	Beispielwert
`string`	Freitext, keine Validierung. Standardtyp.	`"Quartalsbericht"`
`number`	Zahl, optional mit `options.min` / `options.max`.	`"2026"`
`boolean`	Wahrheitswert.	`"false"`
`select`	Auswahl aus `options.items` mit `value` und `label`.	`"EU"`
`date`	Datum im Format `YYYY-MM-DD`.	`"2026-01-01"`
`date_time`	Zeitpunkt nach ISO 8601.	`"2026-01-01T00:00:00Z"`

Jede Parameterdefinition kennt außerdem:

Feld	Beschreibung
`name`	Pflichtfeld. Der Name, den du als `${NAME}` verwendest. Groß-/Kleinschreibung zählt.
`description`	Freitext. Erscheint als Beschriftung im Live-Report (Kapitel 209).
`default`	Wert, der greift, wenn nichts übergeben wurde.
`required`	`true` erzwingt eine Übergabe – ohne `default` bricht der Build sonst ab.
`options`	Typabhängig: `min` / `max` bei `number`, `items` bei `select`.

`number` mit `min` / `max` ist mehr als Kosmetik: Der Build bricht bei einem Wert außerhalb des Bereichs ab, statt einen unsinnigen Bericht zu erzeugen.

---

05 Rangfolge: Was gewinnt?

Für jeden Parameter wird der Wert in dieser Reihenfolge bestimmt (der erste Treffer gewinnt):

1. Der explizite Wert in `params:` des `ReportArtefact`.
2. Der `default` aus der Parameterdefinition.
3. Eine Umgebungsvariable gleichen Namens.

Fehlt danach immer noch ein Wert und ist der Parameter `required`, bricht der Build mit einer klaren Fehlermeldung ab.

Punkt 3 ist praktisch für CI-Pipelines – Werte lassen sich von außen setzen, ohne Manifeste anzufassen:

```
layoutPages:
  - page: regionsseite
    params:
      REGION: ${DEFAULT_REGION}      # aus der Umgebung
      JAHR: ${REPORT_YEAR:2026}      # mit Fallback, falls nicht gesetzt
```

```
DEFAULT_REGION=US bino build --artefact regionsbericht
```

> **Achtung:** Gib Umgebungsvariablen in Preview-Manifesten immer einen Fallback (`${VAR:default}`). In `bino preview` warnt BinoBI bei einer unaufgelösten Variable nur und setzt einen leeren String ein – in `bino build` ist es ein harter Fehler. Ein Bericht, der lokal läuft und in der Pipeline abbricht, kostet unnötig Zeit.

---

06 Parameter auf referenzierten Komponenten

Nicht nur Seiten können Parameter haben. Eigenständige Komponenten-Dokumente – `Text`, `Table`, `ChartStructure`, `ChartTime`, `Tree`, `Grid`, `LayoutCard`, `Image` – deklarieren sie genauso in `metadata.params`. Wer sie per `ref` einbindet, übergibt Werte über `params`:

```
apiVersion: bino.bi/v1alpha1
kind: Text
metadata:
  name: regionskommentar
  params:
    - name: REGION
      type: string
      default: "EU"
spec:
  value: "Kommentar zur Region ${REGION}"
---
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: kommentarseite
spec:
  children:
    - kind: Text
      ref: regionskommentar
      params:
        REGION: "US"
```

Rangfolge, Defaults und `${NAME_LABEL}` verhalten sich exakt wie bei Seitenparametern. Besonders wichtig wird das bei Paketen aus einer Registry (Kapitel 214): Ein veröffentlichtes Paket ist genau dann wiederverwendbar, wenn es seine Variablen sauber als Parameter deklariert.

---

07 Parameter in `bino preview`

Ein Punkt, der regelmäßig für Verwirrung sorgt:

> **`bino preview` rendert alle LayoutPages ohne Parameter.**

Die Vorschau kennt die Seite, aber nicht den Kontext eines Artefakts – sie weiß also nicht, ob sie `EU` oder `US` einsetzen soll. Platzhalter ohne `default` bleiben in der Preview darum sichtbar leer.

Zwei Wege führen weiter:

- **`default` setzen.** Jeder Parameter mit sinnvollem Default macht die Seite auch ohne Kontext ansehnlich – das ist der Normalfall und kostet nichts.
- **`bino serve` mit einem `LiveReportArtefact`.** Dort werden Parameter zu echten Bedienelementen, die per Query-Parameter gesetzt werden (Kapitel 209).

Für den finalen Blick gilt ohnehin: `bino build` erzeugt das, was die Empfänger:innen sehen.

---

08 Wann Parameter – und wann nicht

Parameter sind das richtige Werkzeug, wenn sich Seiten **strukturell gleichen** und nur in Werten unterscheiden. Sie sind das falsche Werkzeug, wenn Seiten unterschiedlich *aufgebaut* sind – eine Seite mit zehn `boolean`-Parametern, die halbe Komponentenbäume ein- und ausschalten, ist schwerer zu pflegen als zwei getrennte Seiten.

Eine brauchbare Faustregel aus der Praxis:

- Bis ca. **fünf Parameter** pro Seite bleibt der Zusammenhang gut lesbar.
- Werden es mehr, ist das meist ein Hinweis, die Seite thematisch zu teilen – etwa in eine geografische, eine zeitliche und eine produktbezogene Seite.
- Was die *Sichtbarkeit* ganzer Manifeste steuert, gehört nicht in Parameter, sondern in **Constraints** (Kapitel 105).

---

09 Typische Fehler

Symptom	Ursache	Lösung
`${REGION}` steht wörtlich im PDF	Name stimmt nicht exakt (Groß-/Kleinschreibung) oder ist nicht deklariert	Namen in `metadata.params` und `params:` abgleichen
Build bricht mit „missing required parameter" ab	`required: true` ohne `default` und ohne Übergabe	Wert im `ReportArtefact` setzen oder `default` ergänzen
Zahl wird nicht ersetzt	Wert als YAML-Zahl statt als String übergeben	`JAHR: "2026"` statt `JAHR: 2026`
Titel zeigt `EU` statt `Europa`	`${REGION}` statt `${REGION_LABEL}` verwendet	Im Titel das Label, im Filter den Wert benutzen

Symptom	Ursache	Lösung
In der Preview ist alles leer	Preview rendert ohne Parameter	`default` setzen oder `bino serve` mit `LiveReportArtefact`

---

10 Tipps

- **Sprechende Namen** zahlen sich aus: `FISCAL_YEAR` statt `Y`, `BUSINESS_UNIT` statt `BU`. Parameter sind eine Schnittstelle und werden von anderen gelesen.
- **`description` ausfüllen**, sobald die Seite in einem `LiveReportArtefact` landet – der Text wird dort zur Beschriftung des Bedienelements.
- **`select` vor `string`**, wann immer die Menge der Werte bekannt ist: Du bekommst Validierung und `${NAME_LABEL}` geschenkt.
- **Denselben Parameternamen** über Seiten hinweg konsistent halten. Das macht ein `ReportArtefact` mit vielen Seiten deutlich lesbarer.

> Vollständige Referenz: cli.bino.bi/guides/layoutpage-params/
> (https://cli.bino.bi/guides/layoutpage-params/)

201 – Fortgeschrittener Bericht: Mehrseiter mit Tree, Drill-Down und Live-Artefakt

> **Hinweis:** Die in diesem Kapitel gezeigten Component-Attribute sind nicht immer vollständig. Die vollständige und aktuelle Referenz findest du unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).

In Kapitel 101 hast du einen einfachen einseitigen Bericht mit einer Tabelle erstellt. In diesem Kapitel gehen wir einen Schritt weiter: Du baust einen mehrseitigen Bericht mit **Custom-Template-Layout**, **Markdown-Text**, **gestapeltem Zeitreihendiagramm**, **Treiberanalyse-Baum** und **parametrisierten Drill-Down-Seiten**.

Dabei lernst du:

- wie du DataSets mit SQL-Transformationen anlegst, um die Datenhierarchie umzustrukturieren,
- wie du Constraints nutzt, um Vorschau- und Build-Daten zu trennen,
- wie du eine Seite mit `custom-template`-Layout und gemischten Komponenten (Table, Text, ChartTime) gestaltest,
- wie du eine Tree-Komponente für Treiberanalysen konfigurierst,
- wie du wiederverwendbare Komponenten (Components) über `ref` einbindest,
- wie du parametrisierte Seiten erstellst und im ReportArtefact mit verschiedenen Werten einbindest,
- und wie du das alles zu einem mehrseitigen PDF zusammenfügst.

Wir bauen den Beispielbericht `handbuch_examples_bn201` Schritt für Schritt auf.

---

01 Überblick: Was wir bauen

Der fertige Bericht `handbuch_examples_bn201.pdf` besteht aus vier Seiten:

Seite	LayoutPage	Inhalt
1	`bn201_example_table_and_text`	Custom-Grid mit Tabelle, Markdown-Text und gestapeltem Zeitreihendiagramm

Seite	LayoutPage	Inhalt
2	`bn201_example_tree`	Analyse-Baum (Tree) mit verschachtelten Charts und Tabellen
3	`bn201_drill_country` (USA)	Parametrisierte Drill-Down-Seite für die Region USA
4	`bn201_drill_country` (Germany)	Dieselbe Seite, erneut eingebunden mit Region Germany

Dafür benötigen wir:

- **3 DataSets:** `bb_example_dataset`, `bb_example_dataset_4_bn201`, `bb_example_dataset_4_bn201_regions`
- **2 wiederverwendbare Components:** `202_country_chart`, `201_city_table`
- **3 LayoutPages:** `bn201_example_table_and_text`, `bn201_example_tree`, `bn201_drill_country`
- **1 ReportArtefact:** `handbuch_examples_bn201`

Außerdem nutzen wir durchgehend **Constraints**, um den Bericht je nach Modus automatisch anzupassen: In der Vorschau werden gesampelte Testdaten verwendet, im Build die vollständigen Produktionsdaten.

> **Voraussetzung:** Dieses Kapitel setzt voraus, dass du bereits ein initialisiertes Projekt mit der DataSource `bb_example_data` hast (siehe Kapitel 101, Abschnitt 03). Die Beispieldaten enthalten Umsatzdaten mit den Spalten des Standard-Datenschemas (siehe Kapitel 103, Abschnitt 08).

---

02 Die DataSets anlegen

Für den Beispielbericht brauchen wir drei DataSets, die auf dieselbe DataSource `bb_example_data` zugreifen, die Daten aber unterschiedlich aufbereiten. Alle drei nutzen **Constraints**, um zwischen Vorschau- und Build-Modus zu unterscheiden (siehe Kapitel 105).

02.01 DataSet 1: `bb_example_dataset` – Pass-Through

Dieses DataSet kennst du bereits aus Kapitel 101, Abschnitt 04. Es reicht die Daten unverändert weiter. Für den fortgeschrittenen Bericht erweitern wir es um Constraints, damit in der Vorschau die gesampelten Daten und im Build die vollständigen Daten verwendet werden.

Erstelle die Datei `datasets/bb_example_dataset_all.yaml`:

```
# Vorschau: gesampelte Daten für schnelles Iterieren
---
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: bb_example_dataset
  constraints:
    - mode == preview
  description: Alle Umsatzdaten
spec:
  query: >-
    SELECT * FROM bb_example_data_preview
  dependencies:
    - bb_example_data_preview


# Build: vollständiger Datensatz
---
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: bb_example_dataset
  constraints:
    - mode != preview
  description: Alle Umsatzdaten
spec:
  query: >-
    SELECT * FROM bb_example_data
  dependencies:
    - bb_example_data
```

Das Muster – **zwei Dokumente mit demselben `metadata.name` und sich gegenseitig ausschließenden Constraints** – kennst du aus Kapitel 105, Abschnitt 04. In der Vorschau wird die gesampelte DataSource `bb_example_data_preview` verwendet, im Build die vollständige `bb_example_data`.

02.02 DataSet 2: `bb_example_dataset_4_bn201` – Hierarchie umstrukturieren

Dieses DataSet bereitet die Daten für die Tabelle auf Seite 1 und die ChartStructure-Knoten im Baum auf Seite 2 auf. Es **mappt die Datenhierarchie um**: Was in den Rohdaten `setname` heißt,

wird hier zu `rowGroup` (die übergeordnete Gruppierung), `rowGroup` wird zu `category` und `category` zu `subCategory`.

Diese Umstrukturierung ist notwendig, weil die in BinoBI CLI verwendete Template Engine einem **data-driven Ansatz** folgt. Die Visualisierung erwartet kein beliebig geformtes Datenmodell, sondern ein DataSet mit einer bestimmten semantischen Struktur. Welche Ebene als `rowGroup`, `category` oder `subCategory` vorliegt, steuert direkt, wie Hierarchien in Tabellen, Struktur-Charts und anderen Komponenten dargestellt werden. Wenn du also für eine Visualisierung einen anderen Fokus setzen oder eine Hierarchie anders schneiden willst, reicht es in der Regel nicht, nur die Komponente umzuschalten: Du erstellst stattdessen ein neues DataSet, das dieselben Rohdaten in die für diesen Use Case passende Form bringt.

Das ist bewusst leichtgewichtig. In der Vorschau wird nur die DataSource im Speicher gecacht. Mehrere DataSets können deshalb parallel auf dieselbe DataSource zugreifen, ohne dass die Rohdaten im Browser mehrfach dupliziert oder zusätzlich im Session-Cache gespeichert werden müssen. Zusätzliche DataSets bedeuten hier also vor allem eine andere Sicht auf dieselbe Datenbasis, nicht eine redundante Datenhaltung.

Erstelle die Datei `datasets/bb_example_dataset_4_bn201.yaml`:

```
# Vorschau: gesampelte Daten für schnelles Iterieren
---
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: bb_example_dataset_4_bn201
  constraints:
    - mode == preview
  description: Alle Umsatzdaten
spec:
  query: >-
    SELECT
    'example' AS setname,
    date,
    operation,
    setname as rowGroup,
    CASE setname
      WHEN 'asia' THEN 10
      WHEN 'australia' THEN 20
      WHEN 'europe' THEN 30
      WHEN 'northamerica' THEN 40
      ELSE NULL
```

```
    END AS rowGroupIndex,
    rowGroup as category,
    rowGroupIndex as categoryIndex,
    category as subCategory,
    categoryIndex as subCategoryIndex,
    columnGroup,
    columnGroupIndex,
    columnSubGroup,
    columnSubGroupIndex,
    pp1, ac1, fc1, pl1,
    pp2, ac2, fc2, pl2,
    pp3, ac3, fc3, pl3,
    pp4, ac4, fc4, pl4
    FROM bb_example_data
  dependencies:
    - bb_example_data_preview


# Build: vollständiger Datensatz
---
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: bb_example_dataset_4_bn201
  constraints:
    - mode != preview
  description: Alle Umsatzdaten
spec:
  query: >-
    SELECT
    'example' AS setname,
    date,
    operation,
    setname as rowGroup,
    CASE setname
      WHEN 'asia' THEN 10
      WHEN 'australia' THEN 20
      WHEN 'europe' THEN 30
      WHEN 'northamerica' THEN 40
      ELSE NULL
    END AS rowGroupIndex,
```

```
    rowGroup as category,
    rowGroupIndex as categoryIndex,
    category as subCategory,
    categoryIndex as subCategoryIndex,
    columnGroup,
    columnGroupIndex,
    columnSubGroup,
    columnSubGroupIndex,
    pp1, ac1, fc1, pl1,
    pp2, ac2, fc2, pl2,
    pp3, ac3, fc3, pl3,
    pp4, ac4, fc4, pl4
    FROM bb_example_data
  dependencies:
    - bb_example_data
```

Was passiert in der SQL-Abfrage?

Die zentrale Transformation ist das **Umordnen der Hierarchieebenen**:

Rohdaten-Spalte	Wird zu	Bedeutung im Bericht
`setname`	`rowGroup`	Übergeordnete Gruppierung (z. B. „europe")
`rowGroup`	`category`	Primäre Dimension (z. B. „Germany")
`category`	`subCategory`	Detail-Dimension (z. B. „Frankfurt")

Zusätzlich wird mit einem `CASE`-Ausdruck ein fester `rowGroupIndex` vergeben, damit die Sortierung der Regionen determiniert ist (asia = 10, australia = 20, europe = 30, northamerica = 40). Die Verwendung von `CASE`-Ausdrücken in DataSets ist in Kapitel 103, Abschnitt 04.01 beschrieben.

02.03 DataSet 3: `bb_example_dataset_4_bn201_regions` – Weltweite Aggregation

Dieses DataSet bereitet die Daten für den Wurzelknoten des Baums auf Seite 2 auf. Es fasst alle Regionen unter einem gemeinsamen `rowGroup`-Wert `'World'` zusammen und macht die einzelnen Regionen (`setname`) zur `category`. Der Wert `'World'` wurde dabei aus fachlichen Gründen gewählt und ist technisch gesehen frei wählbar – innerhalb des Berichts wird nicht nach `setname` gefiltert.

Erstelle die Datei `datasets/bb_example_dataset_4_bn201_regions.yaml`:

```
# Vorschau: gesampelte Daten für schnelles Iterieren
---
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: bb_example_dataset_4_bn201_regions
  constraints:
    - mode == preview
  description: Alle Umsatzdaten
spec:
  query: >-
    SELECT
    'example' AS setname,
    date,
    operation,
    'World' as rowGroup,
    1 as rowGroupIndex,
    setname as category,
    CASE setname
      WHEN 'asia' THEN 10
      WHEN 'australia' THEN 20
      WHEN 'europe' THEN 30
      WHEN 'northamerica' THEN 40
      ELSE NULL
    END AS categoryIndex,
    rowGroup as subCategory,
    rowGroupIndex as subCategoryIndex,
    columnGroup,
    columnGroupIndex,
    columnSubGroup,
    columnSubGroupIndex,
    pp1, ac1, fc1, pl1,
    pp2, ac2, fc2, pl2,
    pp3, ac3, fc3, pl3,
    pp4, ac4, fc4, pl4
    FROM bb_example_data
  dependencies:
    - bb_example_data_preview


# Build: vollständiger Datensatz
```

```
---
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: bb_example_dataset_4_bn201_regions
  constraints:
    - mode != preview
  description: Alle Umsatzdaten
spec:
  query: >-
    SELECT
    'example' AS setname,
    date,
    operation,
    'World' as rowGroup,
    1 as rowGroupIndex,
    setname as category,
    CASE setname
      WHEN 'asia' THEN 10
      WHEN 'australia' THEN 20
      WHEN 'europe' THEN 30
      WHEN 'northamerica' THEN 40
      ELSE NULL
    END AS categoryIndex,
    rowGroup as subCategory,
    rowGroupIndex as subCategoryIndex,
    columnGroup,
    columnGroupIndex,
    columnSubGroup,
    columnSubGroupIndex,
    pp1, ac1, fc1, pl1,
    pp2, ac2, fc2, pl2,
    pp3, ac3, fc3, pl3,
    pp4, ac4, fc4, pl4
    FROM bb_example_data
  dependencies:
    - bb_example_data
```

Der Unterschied zum vorherigen DataSet: Hier wird `'World'` als fester `rowGroup`-Wert verwendet und `setname` (die Regionen) wird zur `category`. So entsteht eine Ansicht, die alle

Regionen als gleichwertige Kategorien unter einem Dach zeigt – ideal für den Wurzelknoten einer Treiberanalyse.

02.04 Warum drei DataSets?

Jedes DataSet bildet eine andere **Perspektive** auf dieselben Rohdaten ab:

DataSet	Perspektive	Verwendet in
`bb_example_dataset`	Rohdaten unverändert	Tree-Blattknoten, Drill-Down
`bb_example_dataset_4_bn201`	Region → Land → Stadt	Tabelle (Seite 1), Tree-Mitte
`bb_example_dataset_4_bn201_regions`	World → Region (aggregiert)	Tree-Wurzelknoten

Visualisierungskomponenten können die Daten anschließend per `filter`-Attribut auf den gewünschten Ausschnitt einschränken (siehe Abschnitt 05.02).

---

03 Seite 1: Tabelle, Text und Zeitreihe mit Custom-Template

Die erste Seite zeigt drei Komponenten in einem freien Raster-Layout: eine Tabelle links, einen Markdown-Text rechts oben und ein gestapeltes Zeitreihendiagramm rechts unten.

03.01 Custom-Template-Layout definieren

Anstelle eines der festen Layouts (`split-horizontal` , `2x2` etc., siehe Kapitel 104, Abschnitt 06) verwenden wir `custom-template`. Das Attribut `pageCustomTemplate` definiert ein CSS-Grid mit benannten Zellen:

```
pageLayout: custom-template
pageCustomTemplate: |
  "A A A A A B B"
  "A A A A A C C"
```

Dieses Grid erzeugt zwei Zeilen und sieben Spalten. Die Zelle **A** erstreckt sich über fünf Spalten und beide Zeilen (die Tabelle), **B** und **C** teilen sich die rechten zwei Spalten (Text oben, Chart unten). Zellen mit demselben Namen werden im Layout jeweils als eine zusammenhängende Fläche interpretiert. Im Beispiel werden die Spalten damit im Verhältnis 5:2 aufgeteilt: **A** nimmt

fünf Breiteneinheiten ein, **B** und **C** jeweils zwei. Die Zellnamen sind frei wählbar, solange sie aus einem zusammenhängenden String ohne Leerzeichen bestehen.

> **Zur Erinnerung:** Die Kind-Komponenten werden von oben links nach unten rechts den Zellen zugeordnet. In diesem Beispiel gilt also: erstes Kind → A, zweites Kind → B, drittes Kind → C. Weitere Details findest du in Kapitel 104, Abschnitt 06.03.

03.02 Das vollständige LayoutPage-Manifest

Erstelle die Datei `pages/examples/bn201_example_table_and_text.yaml`:

```
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: bn201_example_table_and_text
  description: Table
spec:
  pageOrientation: landscape
  pageFormat: a4
  pageLayout: custom-template
  pageCustomTemplate: |
    "A A A A A B B"
    "A A A A A C C"
  titleBusinessUnit: "BinoBI – Example Page with Table, Timeline and Text Compor
  titleMeasures:
    - name: "Revenue"
      unit: "kEUR"
  titleDateStart: 2020-01-01
  titleDateEnd: 2026-12-31
  titleScenarios: [ac1, pl1, fc1]
  titleDateFormat: year
  titleDateLink: interval
  titleOrder: category
  titleOrderDirection: asc
  children:
    - kind: Table
      spec:
        scale: none
        dataset: bb_example_dataset_4_bn201
        sumTitle: All Shop Locations
        level: rowGroup
```

```
      order: categoryindex
      orderDirection: asc
      measureScale: k
      measureType: currency
      measureUnit: EUR
      limit: 30
      unitScaling: 0.0002
      percentageScaling: 2
      scenarios: [ac1, pl1, fc1]
      variances: [dac1_fc1_pos, drac1_fc1_pos, dac1_pl1_pos, drac1_pl1_pos]
      barColumns: [ac1, drac1_pl1_pos]
      barColumnWidth: w10
      grouped: true
      showGroupTitle: true
      showMeasureScale: true
      dataFormat: decimal
      dataFormatDigitsDecimal: 1
      dataFormatDigitsPercent: 2
      type: sum
      categoryWidth: w10

  - kind: Text
    spec:
      scale: none
      value: |
        ## Important Text

        Ipsum dolor sit amet, consectetur adipiscing elit. Sed do eiusmod
        tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
        veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
        ea commodo consequat.

        ### Markdown Elements Guide

        *This text will be italic*   **This text will be bold**
        _You **can** combine them_

         ### Ordered

        1. Item 1
        2. Item 2
```

```
            1. Item 2a
            2. Item 2b

        Ipsum dolor sit amet, consectetur adipiscing elit. Sed do eiusmod
        tempor incididunt ut labore et dolore magna aliqua.

      dataset: bb_example_dataset_4_bn201

  - kind: ChartTime
    metadata:
      name: Example Table
    spec:
      dataset: bb_example_dataset_4_bn201
      chartMode: bar
      dateInterval: year
      measureScale: k
      measureUnit: "EUR"
      percentageScaling: 1
      unitScaling: auto

      stack:
        by: scenarios
        mode: absolute
        order: dataset
```

03.03 Neue Konzepte auf dieser Seite

Die Text-Komponente

Die `Text`-Komponente rendert **Markdown** direkt auf der Berichtsseite. Du schreibst den Inhalt im `value`-Feld mit mehrzeiligem YAML-String (`|`). Unterstützt werden u. a. Überschriften, Fettdruck, Kursivschrift und nummerierte Listen.

Feld	Beschreibung
`value`	Markdown-Text, der gerendert wird.
`dataset`	Optionale Referenz auf ein DataSet – ermöglicht es, Datenwerte in den Text einzubetten.
`scale`	Skalierung der Komponente. `none` bedeutet: keine automatische Skalierung.

> **Tipp:** Die vollständige Attribut-Referenz für die Text-Komponente findest du unter cli.bino.bi/reference/text/ (https://cli.bino.bi/reference/text/).

ChartTime mit gestapelten Balken (`stack`)

Die `ChartTime` -Komponente zeigt Zeitreihen. Neu in diesem Beispiel ist die **Stacked-Chart-Konfiguration** über das `stack` -Objekt:

Feld	Beschreibung
`chartMode`	Darstellungsart: `bar` (Balken) oder `line` (Linie).
`dateInterval`	Aggregationsintervall der Zeitachse: `year` , `quarter` , `month` , `week` , `day` .
`stack.by`	Wonach gestapelt wird: `scenarios` (Szenarien übereinander) oder `dimensions` (Kategorien übereinander).
`stack.mode`	Stapelmodus: `absolute` (absolute Werte), `relative` (normalisiert auf 100%), `absolute-relative` (beides).
`stack.order`	Reihenfolge der Stapel: `dataset` (Reihenfolge aus dem DataSet).

> **Tipp:** Die vollständige Attribut-Referenz für ChartTime findest du unter cli.bino.bi/reference/chart-time/ (https://cli.bino.bi/reference/chart-time/).

Gruppierte Tabelle (`grouped: true`)

Durch `grouped: true` werden die Tabellenzeilen nach der `rowGroup` -Dimension gruppiert. Jede Gruppe erhält eine eigene Überschrift (wenn `showGroupTitle: true`) und eine Zwischensumme. Das `level: rowGroup` bedeutet, dass die Tabelle auf der Ebene der `rowGroup` -Dimension aggregiert – die einzelnen Kategorien werden innerhalb der Gruppen dargestellt.

Mehrere Varianzen und Bar-Spalten

Diese Tabelle zeigt vier Varianzen gleichzeitig:

Varianz	Bedeutung
`dac1_fc1_pos`	Absolute Abweichung AC vs. FC (positiv = gut)
`drac1_fc1_pos`	Relative Abweichung AC vs. FC (positiv = gut)
`dac1_pl1_pos`	Absolute Abweichung AC vs. Plan (positiv = gut)
`drac1_pl1_pos`	Relative Abweichung AC vs. Plan (positiv = gut)

Mit `barColumns: [ac1, drac1_pl1_pos]` werden die Spalten `ac1` und `drac1_pl1_pos` als Inline-Balkendiagramme dargestellt. Das Attribut `barColumnWidth: w10` steuert die Breite dieser Balkenspalten. Das Varianzschema ist in Kapitel 101, Abschnitt 05.04 erklärt.

---

04 Seite 2: Die Tree-Komponente (Treiberanalyse)

Die zweite Seite zeigt einen **Treiberanalyse-Baum** – eine hierarchische Darstellung, die vom Gesamtwert (World) über Regionen (Europe, Asia, ...) bis zu einzelnen Ländern (Germany, France, ...) aufgliedert.

04.01 Constraints auf Seitenebene

Diese Seite verwendet einen Constraint, um sie **nur im Build** einzubinden und in der Vorschau auszublenden. Das ist nützlich, wenn eine Seite noch in Entwicklung ist oder in der Vorschau zu viele Ressourcen benötigt:

````
metadata:
  name: bn201_example_tree
  description: Wird nicht in Preview angezeigt, da es noch fehler
  metadata:
    constraints:
      - mode != preview
``` Constraints auf Seitenebene funktionieren nach demselben Prinzip wie bei Dat

### 04.02 Wiederverwendbare Components mit `ref`

Bevor wir die Tree-Seite anlegen, erstellen wir zwei **wiederverwendbare Kompone

**Component 1: `202_country_chart`** – Ein ChartStructure für Regionsübersichter

Erstelle die Datei `components/202/202_country_chart.yaml`:

```yaml
apiVersion: bino.bi/v1alpha1
kind: ChartStructure
metadata:
  name: 202_country_chart
spec:
  dataset: bb_example_dataset_4_bn201
  level: category
````

```
  order: ac1
  orderDirection: desc
  showMeasureScale: true
  measureScale: k
  measureUnit: EUR
  limit: 3
  unitScaling: 0.0001
  scenarios: [ac1, pp1]
```

Component 2: `201_city_table` – Eine Tabelle für Länder-Detailansichten.

Erstelle die Datei `components/202/202_city_table.yaml`:

```
apiVersion: bino.bi/v1alpha1
kind: Table
metadata:
  name: 201_city_table
spec:
  dataset: bb_example_dataset_4
  level: subcategory
  order: ac1
  orderDirection: desc
  showMeasureScale: true
  measureScale: k
  measureUnit: EUR
  limit: 5
  unitScaling: 0.001
  type: sum
  scenarios: [ac1, pp1]
```

Diese Components werden in den Baumknoten über das `ref`-Attribut eingebunden. BinoBI übernimmt dabei die gesamte Konfiguration der referenzierten Komponente als Basis. Attribute, die du im jeweiligen Knoten direkt angibst (z. B. `dataset`, `filter`, `chartTitle`), **überschreiben** die Werte aus der Vorlage.

04.03 Die Tree-Komponente im Detail

Erstelle die Datei `pages/examples/bn201_example_tree.yaml`:

```
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
```

```
metadata:
  name: bn201_example_tree
  description: Wird nicht in Preview angezeigt, da es noch fehler
  metadata:
    constraints:
      - mode != preview
spec:
  pageOrientation: landscape
  pageFormat: a4
  pageLayout: full
  titleBusinessUnit: "BinoBI – Example Tree Page"
  titleDateStart: 2026-01-01
  titleDateEnd: 2026-12-31
  titleMeasures:
    - name: "Revenue"
      unit: "kEUR"
  titleScenarios: [ac1, pp1]
  titleDateFormat: year
  titleDateLink: end
  titleOrder: category
  titleOrderDirection: asc
  children:
    - kind: Tree
      metadata:
        name: roi_driver_tree
        description: Return on Investment decomposition
      spec:
        direction: ltr
        edgeStyle: orthogonal
        levelSpacing: 20
        nodeSpacing: 10
        showOperators: true
        edges:
          - { from: "rowGroup", to: "category_1", operator: "none" }
          - { from: "rowGroup", to: "category_2", operator: "none" }
          - { from: "rowGroup", to: "category_3", operator: "none" }
          - { from: "rowGroup", to: "category_4", operator: "none" }
          - { from: "category_1", to: "subCategory_1", operator: "none" }
          - { from: "category_1", to: "subCategory_1b", operator: "none" }
          - { from: "category_2", to: "subCategory_2", operator: "none" }
          - { from: "category_2", to: "subCategory_2b", operator: "none" }
```

```
    - { from: "category_3", to: "subCategory_3", operator: "none" }
    - { from: "category_4", to: "subCategory_4", operator: "none" }
  nodes:
    - id: rowGroup
      kind: ChartStructure
      ref: 202_country_chart
      spec:
        dataset: bb_example_dataset_4_bn201_regions
        chartTitle: Worldwide
    - id: category_1
      kind: ChartStructure
      ref: 202_country_chart
      optional: true
      spec:
        dataset: bb_example_dataset_4_bn201
        chartTitle: "Europe"
        filter: rowGroup IN ('europe')
    - id: category_2
      kind: ChartStructure
      ref: 202_country_chart
      optional: true
      spec:
        dataset: bb_example_dataset_4_bn201
        chartTitle: "Asia"
        filter: rowGroup IN ('asia')
    - id: category_3
      kind: ChartStructure
      ref: 202_country_chart
      optional: true
      spec:
        dataset: bb_example_dataset_4_bn201
        chartTitle: "North America"
        filter: rowGroup IN ('northamerica')
    - id: category_4
      kind: ChartStructure
      ref: 202_country_chart
      optional: true
      spec:
        dataset: bb_example_dataset_4_bn201
        chartTitle: "Australia"
        filter: rowGroup IN ('australia')
```

```
      - id: subCategory_1
        kind: Table
        ref: 201_city_table
        optional: true
        spec:
          dataset: bb_example_dataset
          sumTitle: "Germany"
          filter: rowGroup IN ('Germany')
      - id: subCategory_1b
        kind: Table
        ref: 201_city_table
        optional: true
        spec:
          dataset: bb_example_dataset
          sumTitle: "France"
          filter: rowGroup IN ('France')
      - id: subCategory_2
        kind: Table
        ref: 201_city_table
        optional: true
        spec:
          dataset: bb_example_dataset
          sumTitle: "China"
          filter: rowGroup IN ('China')
      - id: subCategory_2b
        kind: Table
        ref: 201_city_table
        optional: true
        spec:
          dataset: bb_example_dataset
          sumTitle: "Japan"
          filter: rowGroup IN ('Japan')
      - id: subCategory_3
        kind: Table
        ref: 201_city_table
        optional: true
        spec:
          dataset: bb_example_dataset
          sumTitle: "USA"
          filter: rowGroup IN ('USA')
      - id: subCategory_4
```

```
            kind: Table
            ref: 201_city_table
            optional: true
            spec:
              dataset: bb_example_dataset
              sumTitle: "Australia"
              filter: rowGroup IN ('Australia')
```

04.04 Tree-Attribute im Überblick

Die `Tree`-Komponente besteht aus **Knoten** (`nodes`) und **Kanten** (`edges`):

Globale Tree-Attribute

Attribut	Beschreibung
`direction`	Richtung des Baums: `ltr` (links nach rechts), `rtl` (rechts nach links), `ttb` (oben nach unten), `btt` (unten nach oben).
`edgeStyle`	Stil der Verbindungslinien: `orthogonal` (rechtwinklig) oder `straight` (gerade).
`levelSpacing`	Abstand zwischen den Hierarchieebenen in Pixeln.
`nodeSpacing`	Abstand zwischen Knoten auf derselben Ebene in Pixeln.
`showOperators`	Operatoren an den Kanten anzeigen (`true` / `false`).

Edge-Definition

Jede Kante verbindet zwei Knoten über ihre `id`:

```
edges:
  - { from: "rowGroup", to: "category_1", operator: "none" }
```

Feld	Beschreibung
`from`	Die `id` des Elternknotens.
`to`	Die `id` des Kindknotens.
`operator`	Mathematischer Operator an der Kante: `none` , `+` , `-` , `*` , `/` . Wird mit `showOperators: true` angezeigt.

Node-Definition

Jeder Knoten ist eine Visualisierungskomponente mit eigener `id`:

Feld	Beschreibung
`id`	Eindeutiger Bezeichner des Knotens – wird in `edges` als `from` / `to` referenziert.
`kind`	Typ der Komponente: `ChartStructure`, `Table`, `ChartTime` etc.
`ref`	Referenz auf eine wiederverwendbare Komponente (Component). Deren Attribute werden als Basis übernommen.
`optional`	Wenn `true`, wird der Knoten ausgelassen, falls der Filter keine Daten liefert (statt einen Fehler zu werfen).
`spec`	Komponentenspezifische Attribute. Überschreiben die Werte aus `ref`.

> **Tipp:** Die vollständige Attribut-Referenz für die Tree-Komponente findest du unter cli.bino.bi/reference/tree/ (https://cli.bino.bi/reference/tree/).

04.05 Wie `ref` und `filter` zusammenspielen

Am Beispiel des Knotens `category_1`:

```
- id: category_1
  kind: ChartStructure
  ref: 202_country_chart
  optional: true
  spec:
    dataset: bb_example_dataset_4_bn201
    chartTitle: "Europe"
    filter: rowGroup IN ('europe')
```

1. BinoBI lädt die Component `202_country_chart` als Basis (dataset, level, order, scenarios etc.).
2. Die `spec`-Attribute im Knoten überschreiben die Basis: `dataset` wird zu `bb_example_dataset_4_bn201`, `chartTitle` zu “Europe”.
3. Der `filter` schränkt die Daten auf Zeilen ein, bei denen `rowGroup = 'europe'`.
4. `optional: true` bedeutet: Wenn der Filter keine Zeilen liefert, wird der Knoten ausgeblendet statt einen Fehler zu erzeugen.

---

05 Seiten 3 und 4: Parametrisierte Drill-Down-Seite

Die letzten beiden Seiten des Berichts basieren auf einer einzigen **parametrisierten LayoutPage** – `bn201_drill_country`. Dieselbe Seite wird im ReportArtefact zweimal eingebunden, jeweils mit einem anderen Regionswert. So entsteht eine Drill-Down-Ansicht pro Region, ohne dass du für jede Region eine eigene Seite schreiben musst.

05.01 Parameter deklarieren

Die Grundlagen zu Parametern kennst du aus Kapitel 104, Abschnitt 09. In diesem Abschnitt lernst du einen neuen Parametertyp kennen: `select` mit einer vordefinierten Werteliste, aus der der Nutzer auswählen kann.

```
metadata:
  name: bn201_drill_country
  params:
    - name: YEAR
      description: reprting date
      type: date
      required: true
    - name: REGION
      description: department location
      type: select
      required: true
      options:
        items:
          - label: USA
            value: USA
          - label: Germany
            value: Germany
          - label: Frankfurt
            value: Frankfurt
          # ... weitere Regionen
```

Der Parameter `YEAR` ist vom Typ `date`, `REGION` vom Typ `select` mit einer vordefinierten Liste von Werten. In der Vorschau erscheinen diese Parameter als auswählbare Eingabefelder in der Toolbar.

05.02 Parameter in Texten und Filtern verwenden

Parameter werden mit `${PARAMETERNAME}` referenziert – sowohl in Textfeldern als auch in `filter`-Ausdrücken:

```
titleBusinessUnit: ${REGION}
messageText: "... for ${REGION}"
filter: "rowGroup = '${REGION}'"
```

BinoBI ersetzt diese Platzhalter beim Build und beim Rendern der Live-Anwendung durch die jeweils übergebenen Werte.

05.03 Das vollständige LayoutPage-Manifest

Erstelle die Datei `pages/examples/bn201_drill_country.yaml`:

```
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  description: test
  name: bn201_drill_country
  params:
    - description: reprting date
      name: YEAR
      required: true
      type: date
    - description: department location
      name: REGION
      options:
        items:
          - label: USA
            value: USA
          - label: Germany
            value: Germany
          - label: Frankfurt
            value: Frankfurt
          - label: Canada
            value: Canada
          - label: UK
            value: UK
          - label: France
            value: France
          - label: Italy
            value: Italy
          - label: China
```

```
            value: China
          - label: Japan
            value: Japan
          - label: Singapore
            value: Singapore
          - label: Hong Kong
            value: Hong Kong
          - label: Australia
            value: Australia
      required: true
      type: select
spec:
  pageOrientation: landscape
  pageFormat: a4
  titleBusinessUnit: ${REGION}
  i18nNamespace: _system
  titleDateStart: 2026-09-31
  titleDateEnd: 2026-09-31
  titleMeasures:
    - name: "Revenue"
      unit: "kEUR"
  titleScenarios: [ac1, pp1]
  titleDateFormat: quarter
  titleDateLink: end
  titleOrder: category
  titleOrderDirection: asc
  pageLayout: split-horizontal
  footerDisplayPageNumber: true
  footerText: "Internal use only"
  messageText: >-
    This is a sample layout page demonstrating the use of a layout card
    with a table. The table is configured to show the top 10 shops by
    revenue for ${REGION}
  children:
    - kind: LayoutCard
      metadata:
        name: Example Card
      spec:
        title: Example Card Title
        description: >-
          This is an example of a layout card containing text and a table.
```

```
      titleImage: logo
      titleOrder: category
      titleOrderDirection: desc
      titleBusinessUnit: "Top 10 Shops in ${REGION}"
      titleScenarios: [ac1, pp1]
      titleDateFormat: quarter
      titleDateLink: end
      scale: auto
      children:
        - kind: Table
          metadata:
            name: Example Table
          spec:
            dataset: cities_bb_dataset
            sumTitle: Shop in ${REGION}
            level: category
            order: ac1
            orderDirection: desc
            measureScale: k
            measureType: currency
            measureUnit: EUR
            limit: 10
            unitScaling: 0.005
            scenarios: [ac1, pl1]
            variances: [dac1_pp1_pos, drac1_pp1_pos]
            grouped: false
            showGroupTitle: true
            showMeasureScale: true
            barColumns: [dac1_pp1_pos, drac1_pp1_pos]
            percentageScaling: 1
            dataFormat: decimal
            dataFormatDigitsDecimal: 0
            dataFormatDigitsPercent: 1
            type: sum
            categoryWidth: w20
            filter: "rowGroup = '${REGION}'"

  - kind: LayoutCard
    metadata:
      name: Example Card
    spec:
```

```
        title: Example Card Title
        description: >-
          This is an example of a layout card containing text and a table.
        titleImage: logo
        titleOrder: category
        titleOrderDirection: desc
        titleBusinessUnit: "All Shops"
        scale: auto
        titleDateStart: 2020-01-01
        titleDateEnd: 2026-10-31
        titleScenarios: ["ac1", "pl1", "fc1"]
        titleDateFormat: quarter
        titleDateLink: interval
        children:
          - kind: ChartTime
            metadata:
              name: Example Table
            spec:
              dataset: cities_bb_dataset
              level: category
              order: ac1
              orderDirection: desc
              showMeasureScale: true
              percentageScaling: 1
              filter: "rowGroup = '${REGION}'"
```

05.04 Was auf dieser Seite passiert

Die Seite verwendet ein `split-horizontal`-Layout (siehe Kapitel 104, Abschnitt 06) mit zwei LayoutCards:

- **Obere Hälfte:** Eine LayoutCard mit einer Tabelle der Top-10-Shops in der gewählten Region, gefiltert über `filter: "rowGroup = '${REGION}'"`.
- **Untere Hälfte:** Eine LayoutCard mit einem ChartTime-Diagramm, das die Umsatzentwicklung über alle Zeiträume für dieselbe Region zeigt.

Beide LayoutCards haben **eigene Titelzeilen** mit unabhängigen Zeiträumen und Szenarien. Die obere Card zeigt ein Quartal mit AC und PP, die untere zeigt den gesamten Zeitraum 2020–2026 mit AC, PL und FC. Dieses Muster – LayoutCards mit unterschiedlichem Kontext – ist in Kapitel 104, Abschnitt 07.03 beschrieben.

Beachte außerdem die Verwendung von `messageText` mit Parameter-Referenz: Der Hinweistext im Header passt sich automatisch an die jeweils übergebene Region an.

---

06 ReportArtefact und LiveReportArtefact

Im letzten Schritt fasst du alle LayoutPages in einer einzigen Datei zusammen. Diese Datei enthält **zwei YAML-Dokumente** mit unterschiedlichen Aufgaben:

- Das `ReportArtefact` bündelt die LayoutPages zu einem reproduzierbaren PDF.
- Das `LiveReportArtefact` macht denselben Inhalt als navigierbare Web-Anwendung mit Routen und URL-Parametern verfügbar.

Die Grundlagen zum `ReportArtefact` kennst du aus Kapitel 101, Abschnitt 06. Neu in diesem Kapitel ist das `LiveReportArtefact`, das den Bericht zusätzlich interaktiv bereitstellt.

Erstelle die Datei `reports/handbuch_examples_bn201.yaml`:

```
apiVersion: bino.bi/v1alpha1
kind: ReportArtefact
metadata:
  name: handbuch_examples_bn201
  description: Example Report mit constrains
spec:
  format: a4
  orientation: landscape
  language: de
  filename: handbuch_examples_bn201.pdf
  title: Example Report 201
  layoutPages:
    - page: bn201_example_table_and_text
    - page: bn201_example_tree
    - page: bn201_drill_country
      params:
        REGION: USA
        YEAR: 2024-12-31T00:00:00Z
    - page: bn201_drill_country
      params:
        REGION: Germany
        YEAR: 2024-12-31T00:00:00Z

---
```

```
apiVersion: bino.bi/v1alpha1
kind: LiveReportArtefact
metadata:
  name: bn201_live
  description: Homepage Dashboard
spec:
  title: BN201 Dashboard
  routes:
    /:
      artefact: handbuch_examples_bn201

    /drill:
      layoutPages: bn201_drill_country
      queryParams:
        - name: REGION
          type: select
          options:
            items:
              - label: USA
                value: USA
              - label: Germany
                value: Germany
              - label: Canada
                value: Canada
              - label: UK
                value: UK
              - label: France
                value: France
              - label: Italy
                value: Italy
              - label: China
                value: China
              - label: Japan
                value: Japan
              - label: Singapore
                value: Singapore
              - label: Hong Kong
                value: Hong Kong
              - label: Australia
                value: Australia
          default: USA
```

```
      - name: YEAR
        type: date
        default: 2024-12-31T00:00:00Z
```

06.01 Parametrisierte Seiten einbinden

Die zentrale Neuerung gegenüber Kapitel 101: Die Seite `bn201_drill_country` wird **zweimal** eingebunden – einmal mit `REGION: USA` und einmal mit `REGION: Germany`. Aus einem einzigen LayoutPage-Manifest entstehen so zwei vollständig separate PDF-Seiten.

Dabei gilt:

- `REGION` muss einem der in der LayoutPage deklarierten `options.items`-Werte entsprechen (z. B. `USA` oder `Germany`).
- `YEAR` erwartet einen ISO-8601-Datumswert.

Du kannst dieselbe Seite beliebig oft einbinden – zum Beispiel einmal pro Land oder einmal pro Quartal.

06.02 Funktion des LiveReportArtefact

Das `LiveReportArtefact` ist der Einstiegspunkt für eine **interaktive Web-Anwendung**. Im Gegensatz zum `ReportArtefact`, das ein statisches PDF erzeugt, beschreibt das `LiveReportArtefact`, welche Inhalte unter welchen URLs erreichbar sind und wie Parameter zur Laufzeit übergeben werden.

In diesem Beispiel definiert `bn201_live` zwei Routen:

- `/` liefert das gerenderte Artefakt `handbuch_examples_bn201` als Startseite aus.
- `/drill` öffnet die LayoutPage `bn201_drill_country` direkt und macht `REGION` sowie `YEAR` als URL-Parameter steuerbar.

Der Unterschied zum `ReportArtefact` ist damit klar: Während du dort die Parameter fest im Manifest verdrahtest und ein reproduzierbares PDF erhältst, werden sie im `LiveReportArtefact` als Benutzereingaben exponiert – ideal für Drill-Down-Ansichten, Dashboards und alle Berichte, die interaktiv genutzt werden sollen.

---

07 Vorschau und Build

07.01 Vorschau starten

Starte jetzt die Vorschau, um den Bericht erstmals zu sehen:

```
bino preview
```

Du siehst alle Seiten des Berichts direkt im Browser. Beachte dabei:

- Die Tree-Seite (`bn201_example_tree`) wird **nicht** angezeigt, da der Constraint `mode != preview` greift.
- Die parametrisierten Drill-Down-Seiten lassen sich über die Dropdown-Felder in der Toolbar mit verschiedenen Regionen und Datumswerten testen.

Weitere Details zur Vorschau findest du in Kapitel 102.

07.02 Live-Anwendung starten

Um den Bericht als interaktive Web-Anwendung zu starten, verwende:

```
bino serve ––live bn201_live
```

`bino serve` startet einen lokalen Server auf Basis des `LiveReportArtefact` . Mit dem Argument `––live bn201_live` wird das entsprechende `LiveReportArtefact` gesgtartet. In unserem Beispiel `bn201_live` . Die definierten Routen sind direkt im Browser erreichbar:

- `/` zeigt das Artefakt `handbuch_examples_bn201` als Startseite.
- `/drill` öffnet die Drill-Down-Seite `bn201_drill_country` , deren Region und Datum du über die URL oder die UI-Steuerelemente anpassen kannst.

Das `LiveReportArtefact` verbindet so Berichtsinhalte mit Routing, Navigation und interaktiv steuerbaren Parametern.

07.03 PDF erzeugen

Wenn du mit dem Ergebnis in der Vorschau zufrieden bist, erzeuge das finale PDF:

```
bino build
```

Das fertige PDF landet in `dist/handbuch_examples_bn201.pdf` und enthält alle vier Seiten – inklusive der Tree-Seite, die wegen ihres Constraints nur im Build gerendert wird.

---

08 Zusammenfassung

In diesem Kapitel hast du einen mehrseitigen Bericht mit fortgeschrittenen Funktionen gebaut:

1. **DataSets mit SQL-Transformationen** – Umstrukturierung der Datenhierarchie durch Spalten-Aliase und CASE-Ausdrücke (Abschnitt 02).
2. **Constraints für Preview/Build** – Zwei Manifeste mit demselben Namen und sich ausschließenden Constraints trennen Entwicklung und Produktion (Abschnitt 02, Kapitel 105).
3. **Custom-Template-Layout** – Freie Grid-Layouts mit `pageCustomTemplate` und CSS grid-template-areas (Abschnitt 03, Kapitel 104 Abschnitt 06.03).
4. **Text-Komponente** – Markdown-Inhalte direkt auf der Berichtsseite rendern (Abschnitt 03.03).
5. **ChartTime mit Stacking** – Gestapelte Zeitreihendiagramme mit dem `stack`-Objekt (Abschnitt 03.03).
6. **Tree-Komponente** – Hierarchische Treiberanalysen mit Knoten, Kanten und verschiedenen Visualisierungstypen (Abschnitt 04).
7. **Wiederverwendbare Components** – Einmalig definierte Komponenten per `ref` in mehreren Baumknoten einsetzen (Abschnitt 04.02).
8. **Parametrisierte Seiten** – Dieselbe LayoutPage mit `params` für verschiedene Regionen einbinden (Abschnitt 05).
9. **Filter** – Daten innerhalb von Komponenten per `filter`-Attribut auf bestimmte Werte einschränken (Abschnitt 05.02).
10. **LiveReportArtefact** – Berichte als interaktive Anwendung mit Routen und `queryParams` bereitstellen (Abschnitt 06.02).

> **Jetzt bist du dran!** Erweitere den Bericht auf eigene Faust: Binde eine weitere Region als zusätzliche Drill-Down-Seite ein, füge dem Treiberanalyse-Baum eine vierte Hierarchieebene hinzu oder experimentiere mit verschiedenen `stack`-Konfigurationen im ChartTime. Überprüfe deine Änderungen jederzeit mit `bino preview` und erzeuge das finale PDF mit `bino build`.

202 – Die Table-Komponente

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/table (https://cli.bino.bi/reference/table).

Die `Table` ist die meistverwendete Visualisierungskomponente in BinoBI CLI. Sie stellt tabellarische Daten mit Szenarien, Varianzen und optionalen Inline-Balkendiagrammen dar. In Kapitel 201 hast du die Tabelle bereits in verschiedenen Kontexten gesehen – auf Seite 1 als gruppierte Übersichtstabelle, in den Tree-Blattknoten als Detail-Tabelle und auf der parametrisierten Drill-Down-Seite. Dieses Kapitel erklärt die Komponente im Detail.

Du lernst:

- welche Attribute die Table-Komponente steuern,
- wie du Hierarchieebenen, Sortierung und Limits konfigurierst,
- wie du Szenarien und Varianzen auswählst und das Varianz-Namensschema liest,
- wie du Inline-Balkendiagramme in Spalten einblendest,
- wie du Gruppierung, Formatierung und Filter einsetzt,
- wie du Drill-Down-Funktionen (`thereof` , `partof` , `columnthereof`) konfigurierst,
- wie du Skalierung und Proportionierung steuerst,
- und wie du eine Table als wiederverwendbare Component definierst.

---

01 Minimalkonfiguration

Eine Tabelle braucht mindestens ein `dataset` :

```
- kind: Table
  spec:
    dataset: bb_example_dataset
```

BinoBI rendert dann eine Standardtabelle mit allen Daten aus dem DataSet. In der Praxis wirst du fast immer weitere Attribute setzen.

---

02 Attribute im Überblick

02.01 Datenanbindung, Hierarchie und Sortierung

Attribut	Typ	Standard	Beschreibung
`dataset`	string	–	Name des DataSets (Pflichtfeld).
`filter`	string	`""`	SQL-`WHERE`-Fragment, das die Zeilen vor der Darstellung einschränkt (z. B. `"rowGroup = 'europe'"`). Unterstützt Template-Parameter wie `${REGION}`.
`order`	string	`category`	Sortierschlüssel. Mögliche Werte: `category`, `categoryindex`, `rowgroup`, `rowgroupindex`, `ac1` – `ac4`, `fc1` – `fc4`, `pp1` – `pp4`, `pl1` – `pl4`, `inherited-closest`, `inherited-page`. Bei Szenario-Namen muss das jeweilige Szenario auch in `scenarios` enthalten sein.
`orderDirection`	string	`asc`	Sortierrichtung: `asc` (aufsteigend), `desc` (absteigend). Auch `inherited-closest` und `inherited-page` möglich.
`limit`	integer	`0`	Maximale Zeilen pro Gruppe. Überschüssige Zeilen werden als **REST** zusammengefasst. `0` bedeutet unbegrenzt.
`type`	string	`list`	Tabellentyp (siehe Tabelle unten).

Tabellentypen (`type`)

Wert	Verhalten
`list`	Einfache Liste ohne Gesamtsumme (Standard).
`sum`	Liste mit Gesamtsumme am Ende. Bei leerem `sumTitle` erscheint dort das IBCS-Summensymbol ❖.
`opt`	Wie `sum`, zeigt aber zusätzlich die +/−-Vorzeichen aus der DataSource an (geeignet für Bilanzen/Kontenrahmen).
`sumnototal`	Wie `sum`, aber ohne Gesamtsummenzeile.
`optnototal`	Wie `opt`, aber ohne Gesamtsummenzeile.

02.02 Szenarien und Varianzen

Attribut	Typ	Standard	Beschreibung
`scenarios`	string / array	–	Szenarien, die als Spalten dargestellt werden. Werte: `ac1` – `ac4`, `fc1` – `fc4`, `pp1` – `pp4`, `pl1` – `pl4`, `auto`, `inherited-closest`, `inherited-page`. `auto` erkennt Szenarien automatisch aus dem DataSet (ohne Warnung). Werden `scenarios` und `variances` leer gelassen, erfolgt Auto-Erkennung mit Warnung.
`variances`	string / array	–	Varianz-Spalten nach dem Muster `d<szenarioB>_<szenarioA>_[pos|neg|neu]` (siehe Abschnitt 03). Auch `inherited-closest` und `inherited-page` möglich.

02.03 Maßeinheiten und Skalierung

Attribut	Typ	Standard	Beschreibung
`measureScale`	string	`GREATEST`	SI-Präfix für die Datenskalierung. `GREATEST` wählt automatisch die größte passende Einheit, `LEAST` die kleinste. Weitere Werte: `_` (keine), `k` (10^3), `M` (10^6), `G` (10^9), `T` (10^{12}), `P` (10^{15}), `E` (10^{18}), `Z`, `Y` sowie die Kleinpräfixe `m`, `µ`, `n`, `p`, `f`, `a`, `z`, `y`.
`measureType`	string	`volume`	Art der Maßeinheit: `volume` oder `currency`. **Hinweis:** Hat aktuell keine visuelle Auswirkung; Standardeinstellung beibehalten.
`measureUnit`	string	`""`	Einheitenbezeichnung in Spaltenköpfen (z. B. `EUR`, `USD`, `Stk`).
`showMeasureScale`	bool	`false`	Zeigt den Skalierungspräfix in der Tabellenüberschrift an.
`unitScaling`	number	`100`	Dateneinheiten je 1 em Balkenlänge für absolute Varianz-Balken (`d_`-Präfix-Spalten). Map-Skala: ein größerer Wert ergibt kürzere Balken.
`percentageScaling`	number	`2`	Prozentpunkte je 1 em Pin-Länge für relative Varianz-Pins (`dr_`-Präfix-Spalten). Ein größerer Wert ergibt kürzere Pins.

02.04 Formatierung

Attribut	Typ	Standard	Beschreibung
`dataFormat`	string	`decimal`	Zahlenformat: `decimal` (Dezimalzahlen) oder `percent` (Prozentwerte, z. B. `0.02` → `2 %`).
`dataFormatDigitsDecimal`	integer	`1`	Nachkommastellen für Dezimalwerte in Tabellenzellen.
`dataFormatDigitsPercent`	integer	`2`	Nachkommastellen für Prozentwerte in Tabellenzellen.

> **Migrationshinweis – `tableTitle` gibt es nicht mehr.** Ältere Projekte verwenden auf `Table` ein Attribut `tableTitle`. Es ist im aktuellen Schema **nicht mehr zulässig** und lässt `bino lint` und `bino build` mit `additional property 'tableTitle' not allowed` abbrechen. Der Nachfolger heißt `sumTitle` – allerdings mit **enger gefasster Bedeutung**: Er beschriftet ausschließlich die Gesamtsummenzeile, nicht die Tabelle als Ganzes.
>
> Beim Umstellen sind zwei Fälle zu unterscheiden:
>
> - **`type: sum` oder `type: opt`:** Reines Umbenennen genügt, die Beschriftung landet wie zuvor in der Summenzeile.
> - **`type: list`, `sumnototal`, `optnototal`:** Diese Typen rendern keine Summenzeile und ignorieren den Wert stillschweigend. Die Lint-Regel `table-sum-title-unused` weist darauf hin. Soll die Tabelle weiterhin eine Überschrift tragen, gehört sie jetzt an das umgebende Layout – etwa `titleBusinessUnit` auf der `LayoutCard` oder `LayoutPage` (Kapitel 104).

02.05 Darstellung und Layout

Attribut	Typ	Standard	Beschreibung
`sumTitle`	string	❖	Beschriftung der **Gesamtsummenzeile** am Ende der Tabelle – **kein** Titel über der Tabelle. Wirkt nur bei `type: sum` und `type: opt`; alle anderen Typen rendern keine Summenzeile und ignorieren den Wert. Bleibt das Feld leer, erscheint das IBCS-Summensymbol ❖.

Attribut	Typ	Standard	Beschreibung
`grouped`	bool	`false`	Gruppiert Zeilen nach der `rowGroup`-Dimension und zeigt den jeweiligen Gruppennamen an.
`showGroupTitle`	bool	`true`	Zeigt Gruppenüberschriften an. Greift nur wenn `grouped: true`.
`categoryWidth`	string	`w15`	Breite der Kategorie-Spalte. Werte: `w5`, `w10`, `w15`, `w20`, `w25`, `w50`, `inf` (Standard = unbegrenzt, maximiert den verfügbaren Platz).
`barColumns`	array	`[]`	Spalten, die statt als Zahl als IBCS-konformes Inline-Balken- bzw. Pin-Diagramm gerendert werden (z. B. `[ac1, drac1_pl1_pos]`). Berücksichtigt werden nur Spalten, die auch in `scenarios` oder `variances` stehen; alles andere wird ohne Warnung ignoriert. Siehe Abschnitt 04.
`barColumnWidth`	string	`w10`	Maximale Breite der Balkenspalten in em. Werte: `w5`, `w10`, `w15`, `w20`, `w25`, `w30`, `w40`, `w60`, `w80`, `w100`.
`scale`	string / number	auto mit Warnung	Proportionale Skalierung der gesamten Tabelle (Schriftgröße, em-Abstände, SVG-Grafiken). Werte: leer/nicht gesetzt = Auto-Skalierung mit `AUTO_scale`-Warnung; `"none"` = keine Skalierung; `"auto"` = Auto-Skalierung ohne Warnung; positive Zahl (z. B. `"0.8"`) = fixer Faktor. Siehe Abschnitt 07.

02.06 Drill-Down

BinoBI unterstützt drei Arten von vertikalem und horizontalem Drill-Down direkt in der Tabelle. Alle drei Attribute sind optional und können kombiniert werden.

Attribut	Typ	Beschreibung
`thereof`	array	**Vertikaler Drill-Down** auf Sub-Category-Ebene. Zeigt für eine bestimmte Kategorie eine eingerückte Unterzeile an. Konfigurierbar über `rowGroup`, `category` und `subCategory`.
`partof`	array	**Vertikaler %-Drill-Down**. Zeigt den relativen Anteil einer Kategorie an ihrer `rowGroup`. Konfigurierbar über `rowGroup` und

Attribut	Typ	Beschreibung
		`category`.
`columnthereof`	array / null	**Horizontaler Drill-Down** für Szenario-Spalten. Blendet für ein bestimmtes Szenario Sub-Gruppen als zusätzliche Spalten ein. Konfigurierbar über `scenario`, `name` und `subGroups`.

```
# Beispiel: Kombinierter Drill-Down
spec:
  thereof:
    - rowGroup: "Revenue"
      category: "Applications"
      subCategory: "Applications-A"
  partof:
    - rowGroup: "Revenue"
      category: "Applications"
  columnthereof:
    - scenario: "ac1"
      name: "DE"
      subGroups: ["Berlin", "Hamburg"]
```

02.07 Internationalisierung

Attribut	Typ	Standard	Beschreibung
`i18nNamespace`	string	`_system`	i18n-Namensraum, in dem die Beschriftungen dieser Komponente aufgelöst werden. Fehlt das Feld, wird der Namensraum vom nächsten übergeordneten Element (Card, Seite, Tree, Grid oder Artefakt) geerbt; Schlüssel, die dort fehlen, fallen auf `_system` zurück.

Details zu Namensräumen und Übersetzungsdateien stehen in Kapitel 208.

02.08 Vererbte Werte (Inherited Props)

Statt `scenarios`, `variances`, `order` und `orderDirection` auf jeder Komponente zu wiederholen, können diese Werte vom übergeordneten `LayoutPage` oder `LayoutCard` geerbt werden. Dazu wird ein spezielles Schlüsselwort als Wert gesetzt:

Schlüsselwort	Quelle
`inherited-closest`	Nächster übergeordneter `LayoutCard` oder `LayoutPage`
`inherited-page`	Nächster übergeordneter `LayoutPage` (überspringt Cards)

Die geerbten Werte stammen aus den Feldern `titleScenarios`, `titleVariances`, `titleOrder` bzw. `titleOrderDirection` des Vorfahren. Wird kein passender Vorfahre gefunden, greift die normale Auto-Erkennung.

```
# Empfänger-Komponente
- kind: Table
  spec:
    dataset: revenue_data
    grouped: true
    type: sum
    scenarios: inherited-page # übernimmt titleScenarios der Seite
    variances: inherited-page # übernimmt titleVariances der Seite
```

---

03 Varianz-Namensschema

Varianzen folgen einem festen Namensmuster: `d[r]<szenarioB>_<szenarioA>_<vorzeichen>`

Teil	Bedeutung
`d`	Absolute Differenz (delta): szenarioB – szenarioA
`dr`	Relative Differenz (delta ratio): (szenarioB – szenarioA) / szenarioA in %
`<szenarioB>`	Vergleichs-Szenario (z. B. `ac1`, `fc1`)
`<szenarioA>`	Basis-Szenario (z. B. `pl1`, `pp1`)
`_pos`	Vorzeichen-Konvention: positive Werte = grün (gut)
`_neg`	Vorzeichen-Konvention: positive Werte = rot (schlecht), z. B. bei Kosten
`_neu`	Neutral: keine farbliche Bewertung

Beispiele:

Varianz-Name	Bedeutung
`dac1_pl1_pos`	AC – PL, absolut, positiv = gut
`drac1_pl1_pos`	(AC – PL) / PL in %, positiv = gut
`dfc1_ac1_neg`	FC – AC, absolut, positiv = schlecht (z. B. Kostenszenario)
`drac1_pp1_pos`	(AC – PP) / PP in %, positiv = gut

> Damit eine Varianz verwendet werden kann, müssen beide beteiligten Szenarien im `scenarios`-Array enthalten sein.

---

04 Inline-Balkendiagramme

Spalten in `barColumns` werden statt als Zahl als IBCS-konformes Inline-Diagramm gerendert. Der Typ des Diagramms richtet sich nach dem Spaltentyp:

Spaltentyp	Diagrammart	Beschreibung
`d_`-Präfix (absolut)	Horizontaler Balken	Balken wächst links (negativ) oder rechts (positiv) von der Mittelachse. Farbe: grün (positiv-gut) oder rot (negativ-schlecht) je nach `_pos` / `_neg`-Suffix.
`dr_`-Präfix (relativ)	Pin-/Dot-Diagramm	Dünne farbige Linie von der Achse mit Punkt am Ende. Bei Extremwerten (> 100 %) erscheint ein Pfeil. Beschriftung kursiv in Prozent.
`ac`, `pp`, `fc`, `pl` (Szenarien)	IBCS-Graustufenbalken	AC: dunkelgraue Vollbalken; PP: hellgraue Vollbalken; FC: dunkelgraue Schraffur; PL: dunkelgrau umrandet.

Die Breite aller Balkenspalten wird über `barColumnWidth` gesteuert (Angabe in em). `unitScaling` (für `d_`) und `percentageScaling` (für `dr_`) legen fest, wie viele Dateneinheiten bzw. Prozentpunkte auf 1 em Balken- bzw. Pin-Länge entfallen. Es ist eine Map-Skala: ein größerer Wert ergibt kürzere Balken. Ist der Wert zu klein, laufen alle Balken in die Begrenzung und werden mit dem Overdrive-Symbol `}}` abgeschnitten.

```
# Varianz-Balken und Szenario-Balken kombiniert
spec:
  scenarios: [ac1, pp1]
  variances: [dac1_pp1_pos, drac1_pp1_pos]
  barColumns: [ac1, dac1_pp1_pos, drac1_pp1_pos]
```

```
  barColumnWidth: w20
  unitScaling: 0.02
  percentageScaling: 1
```

---

05 Beispiel: Gruppierte Tabelle mit Inline-Balken

Dieses Beispiel aus Kapitel 201, Abschnitt 03.02 zeigt eine Tabelle mit allen wichtigen Attributen:

```
- kind: Table
  spec:
    scale: none
    dataset: bb_example_dataset_4_bn201
    sumTitle: All Shop Locations
    level: rowGroup
    order: categoryindex
    orderDirection: asc
    measureScale: k
    measureType: currency
    measureUnit: EUR
    limit: 30
    unitScaling: 0.0002
    percentageScaling: 2
    scenarios: [ac1, pl1, fc1]
    variances: [dac1_fc1_pos, drac1_fc1_pos, dac1_pl1_pos, drac1_pl1_pos]
    barColumns: [ac1, drac1_pl1_pos]
    barColumnWidth: w10
    grouped: true
    showGroupTitle: true
    showMeasureScale: true
    dataFormat: decimal
    dataFormatDigitsDecimal: 1
    dataFormatDigitsPercent: 2
    type: sum
    categoryWidth: w10
```

in kEUR	PL	FC	AC	ΔPL	ΔFC	ΔPL%	ΔFC%
asia			-			-	
Japan	19,1	19,1	19	-0,3	-0,3	-1,60%	-1,70%
Singapore	10,3	10,4	10	-0,1	-0,2	-1,37%	-2,35%
Hong Kong (SAR …	17,2	17,3	17	-0,3	-0,4	-1,82%	-2,09%
China	38,0	38,3	37	-0,6	-0,8	-1,49%	-2,03%
South Korea	12,4	12,4	12	-0,1	-0,1	-0,77%	-0,83%
India	12,4	12,4	12	+0,0	+0,1	+0,28%	+0,74%
UAE	12,4	12,4	12	+0,1	+0,1	+0,92%	+0,47%
asia	**121,9**	**122,3**	**121**	**-1,3**	**-1,7**	***-1,04%***	***-1,36%***
australia			-			-	
Australia	33,7	33,7	33	-0,5	-0,6	-1,58%	-1,65%
New Zealand	12,4	12,4	12	0,0	-0,1	-0,34%	-0,45%
australia	**46,1**	**46,2**	**46**	**-0,6**	**-0,6**	***-1,25%***	***-1,32%***
europe			-			-	
UK	16,8	16,9	17	-0,2	-0,4	-1,18%	-2,12%
France	18,2	18,4	18	-0,3	-0,5	-1,50%	-2,48%
Italy	11,1	11,0	11	-0,2	-0,2	-2,21%	-1,40%
Germany	39,8	39,8	39	-0,5	-0,5	-1,27%	-1,37%
Spain	12,5	12,5	13	+0,1	+0,0	+0,50%	+0,19%
Netherlands	12,4	12,6	12	+0,0	-0,1	+0,14%	-0,76%
Switzerland	12,4	12,5	12	+0,1	0,0	+0,53%	-0,14%
europe	**123,1**	**123,6**	**122**	**-1,1**	**-1,6**	***-0,87%***	***-1,30%***
northamerica			-			-	
USA	83,0	83,6	82	-0,9	-1,5	-1,12%	-1,83%
Canada	14,8	14,8	15	-0,2	-0,2	-1,63%	-1,64%
Mexico	12,4	12,4	12	+0,1	+0,0	+0,58%	+0,31%
northamerica	**110,1**	**110,8**	**109**	**-1,1**	**-1,7**	***-1,00%***	***-1,56%***
All Shop Locations	**401,2**	**402,8**	**397**	**-4,0**	**-5,6**	***-1,00%***	***-1,39%***

Gruppierte Tabelle mit Inline-Balken

05.01 Was hier passiert

1. **Daten:** Das DataSet `bb_example_dataset_4_bn201` liefert eine umstrukturierte Hierarchie (Region → Land → Stadt, siehe Kapitel 201, Abschnitt 02.02).
2. **Typ `sum`:** Jede Gruppe erhält eine Zwischensumme, die Gesamtsumme steht am Ende der Tabelle.

3. **Gruppierung:** `grouped: true` gruppiert die Zeilen nach `rowGroup` (den Regionen). Mit `showGroupTitle: true` erhält jede Gruppe eine Überschrift.
4. **Szenarien:** Drei Spalten – AC (Ist), PL (Plan), FC (Forecast).
5. **Varianzen:** Vier Varianz-Spalten – absolute und relative Abweichungen jeweils für AC vs. FC und AC vs. PL. Alle mit `_pos`, da höhere Umsätze positiv bewertet werden.
6. **Inline-Balken:** Die Spalten `ac1` (Szenario-Balken, IBCS-Graustufen) und `drac1_pl1_pos` (relativer Pin-Balken) werden als Diagramme dargestellt.
7. **Sortierung:** Nach `categoryindex` aufsteigend – die Reihenfolge wird durch den Index im DataSet bestimmt, nicht durch Werte.
8. **Skalierung:** `scale: none` verhindert automatisches Größenanpassen; die Tabelle rendert in der Basis-Schriftgröße.

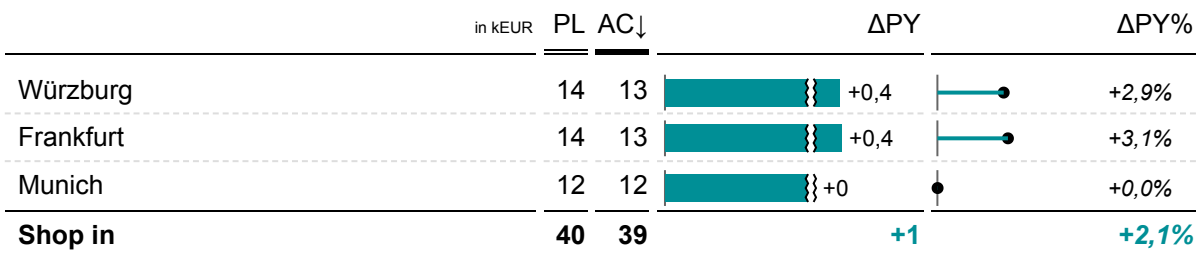
---

06 Beispiel: Gefilterte Detail-Tabelle

Aus der Drill-Down-Seite (Kapitel 201, Abschnitt 05):

```
- kind: Table
  spec:
    dataset: cities_bb_dataset
    sumTitle: Shop in ${REGION}
    level: category
    order: ac1
    orderDirection: desc
    measureScale: k
    measureType: currency
    measureUnit: EUR
    limit: 10
    unitScaling: 0.005
    scenarios: [ac1, pl1]
    variances: [dac1_pp1_pos, drac1_pp1_pos]
    barColumns: [dac1_pp1_pos, drac1_pp1_pos]
    grouped: false
    showGroupTitle: true
    showMeasureScale: true
    percentageScaling: 1
    dataFormat: decimal
    dataFormatDigitsDecimal: 0
    dataFormatDigitsPercent: 1
    type: sum
```

```
    categoryWidth: w20
    filter: "rowGroup = '${REGION}'"
```



in kEUR	PL	AC↓	ΔPY	ΔPY%
Würzburg	14	13	+0,4	+2,9%
Frankfurt	14	13	+0,4	+3,1%
Munich	12	12	+0	+0,0%
Shop in	**40**	**39**	**+1**	**+2,1%**

Gefilterte Detail-Tabelle (filter: rowGroup = 'Germany'

06.01 Filter und Parameter

Das `filter`-Attribut akzeptiert einen AlaSQL-`WHERE`-Ausdruck und schränkt die geladenen Zeilen vor der Darstellung ein. Unterstützte Operatoren sind u. a. `=`, `!=`, `IN (...)`, `LIKE`, `AND`, `OR`. In Kombination mit Seiten-Parametern (hier `${REGION}`) entsteht eine dynamische Tabelle, die je nach übergebenem Wert unterschiedliche Daten zeigt. Das Zusammenspiel von Filtern und Parametern ist in Kapitel 201, Abschnitt 05.02 beschrieben.

---

07 Proportionale Skalierung (`scale`)

Das `scale`-Attribut skaliert die gesamte Tabelle proportional: Schriftgröße, em-basierte Abstände, Rahmenbreiten, SVG-Balken, Pin-Längen, Punkt-Radien und Schraffurmuster skalieren gemeinsam.

Wert	Verhalten
nicht gesetzt / `""`	Auto-Skalierung auf die Breite des übergeordneten Containers. Gibt eine `AUTO_scale`-Warnung mit dem angewendeten Faktor aus.
`"none"`	Keine Skalierung. Die Tabelle rendert in der Basis-Schriftgröße, unabhängig vom verfügbaren Platz.
`"auto"`	Auto-Skalierung auf die Container-Breite, ohne Warnung.
`"0.5"`, `"1.2"` …	Fixer Skalierungsfaktor, der auf die Basis-Schriftgröße angewendet wird.

```
# Auto-Skalierung ohne Warnung (empfohlen für Cards mit variabler Breite)
- kind: Table
```

```
  spec:
    dataset: revenue_by_segment
    scale: auto
    scenarios: [ac1, pp1]
```

```
# Feste Größe – z. B. für PDF-Seiten mit bekanntem Layout
- kind: Table
  spec:
    dataset: revenue_by_segment
    scale: none
    scenarios: [ac1, pp1]
```

---

08 Drill-Down Beispiel: thereof, partof, columnthereof

```
apiVersion: bino.bi/v1alpha1
kind: Table
metadata:
  name: revenue_grouped_table
spec:
  dataset: revenue_detail
  measureScale: M
  measureType: currency
  measureUnit: "EUR"
  grouped: true
  showGroupTitle: true
  categoryWidth: w25
  type: list
  limit: 5
  scenarios: ["ac1", "py1"]
  variances: ["dpy1_ac1_pos"]
  thereof:
    - rowGroup: "Revenue"
      category: "Applications"
      subCategory: "Applications-A"
  partof:
    - rowGroup: "Revenue"
      category: "Applications"
  columnthereof:
```

```
    - scenario: "ac1"
      name: "DE"
      subGroups: ["Berlin", "Hamburg"]
```

Tabelle. revenue_grouped_table

Was die Drill-Down-Attribute bewirken:

- **`thereof`:** Fügt unter der Kategorie „Applications" eine eingerückte Zeile für die Sub-Category „Applications-A" ein, die deren Anteil an der Kategorie zeigt (*davon*-Logik).
- **`partof`:** Fügt neben der Kategorie „Applications" eine %-Spalte ein, die den Anteil an der `rowGroup` „Revenue" darstellt (*Anteil-von*-Logik).
- **`columnthereof`:** Klappt die `ac1`-Spalte horizontal auf und zeigt neben dem Gesamtwert die Werte für die Sub-Gruppen „Berlin" und „Hamburg" als eigene Spalten.

---

09 Table als wiederverwendbare Component

Tabellen-Konfigurationen, die in mehreren Kontexten verwendet werden, kannst du als eigenständige Component-Datei anlegen:

```
# components/202/202_city_table.yaml
apiVersion: bino.bi/v1alpha1
kind: Table
metadata:
  name: 201_city_table
spec:
  dataset: bb_example_dataset_4
  level: subcategory
  order: ac1
  orderDirection: desc
  showMeasureScale: true
  measureScale: k
  measureUnit: EUR
  limit: 5
  unitScaling: 0.001
  type: sum
  scenarios: [ac1, pp1]
```

Diese Component wird dann per `ref` eingebunden – z. B. in einem Tree-Knoten:

```
- id: subCategory_1
  kind: Table
  ref: 201_city_table
  optional: true
  spec:
    dataset: bb_example_dataset
    sumTitle: "Germany"
    filter: rowGroup IN ('Germany')
```

Die `spec`-Attribute im einbindenden Manifest überschreiben die Werte aus der Component. So kannst du eine Basis-Konfiguration definieren und je nach Kontext nur `dataset`, `sumTitle` und `filter` anpassen. Dieses Muster ist in Kapitel 201, Abschnitt 04.02 ausführlich erklärt.

---

06 Tipps

- **Sortierung nach Index vs. Wert:** `order: categoryindex` sortiert nach der Reihenfolge im DataSet, `order: ac1` nach dem Ist-Wert. Für determinierte Reihenfolgen (z. B. Regionen) verwende Index-Spalten.
- **Limit setzen:** Bei großen DataSets verhindert `limit` eine überladene Tabelle. Die Summenzeile berücksichtigt trotzdem alle Daten.
- **Breiten anpassen:** `categoryWidth` und `barColumnWidth` steuern die Proportionen. Experimentiere mit verschiedenen Werten in der Vorschau (`bino preview`).
- **Varianzen weglassen:** Wenn du keine Abweichungen brauchst, setze `variances: []` oder lasse das Attribut weg.

> **Tipp:** Die vollständige Attribut-Referenz für die Table-Komponente findest du unter cli.bino.bi/reference/table/ (https://cli.bino.bi/reference/table/).

---

07 Zusammenfassung

In diesem Kapitel hast du die Table-Komponente im Detail kennengelernt:

1. **Datenanbindung** – `dataset`, `level`, `order`, `limit` und `filter` steuern, welche Daten in welcher Form angezeigt werden.
2. **Szenarien und Varianzen** – `scenarios` und `variances` definieren die Spalten der Tabelle.

3. **Inline-Balken** – `barColumns` und `barColumnWidth` fügen visuelle Indikatoren in die Tabelle ein.
4. **Gruppierung** – `grouped` und `showGroupTitle` erzeugen hierarchische Tabellen mit Zwischensummen.
5. **Formatierung** – `dataFormat`, `dataFormatDigitsDecimal` und `dataFormatDigitsPercent` steuern die Zahlendarstellung.
6. **Wiederverwendung** – Tabellen-Konfigurationen können als Component angelegt und per `ref` in verschiedenen Kontexten eingebunden werden.

> **Jetzt bist du dran!** Erstelle eine gruppierte Tabelle mit drei Szenarien und mindestens einer Varianz. Aktiviere Inline-Balken für die Ist-Spalte und experimentiere mit verschiedenen `categoryWidth`- und `barColumnWidth`-Werten. Prüfe das Ergebnis mit `bino preview`.

203 – Die ChartStructure-Komponente

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/chart-structure/ (https://cli.bino.bi/reference/chart-structure/).

Die `ChartStructure`-Komponente stellt **strukturelle Vergleiche** als horizontales Balkendiagramm dar – z. B. den Umsatz nach Regionen oder Produktgruppen. Im Gegensatz zur `ChartTime` (Kapitel 204), die Daten entlang einer Zeitachse zeigt, vergleicht `ChartStructure` Kategorien innerhalb eines Zeitpunkts oder Zeitraums.

In Kapitel 201 hast du ChartStructure bereits als Knoten im Treiberanalyse-Baum gesehen (Abschnitt 04). Dieses Kapitel erklärt die Komponente im Detail.

Du lernst:

- welche Attribute die ChartStructure-Komponente steuern,
- wie du Hierarchieebene, Sortierung und Limit konfigurierst,
- wie du Szenarien und Varianzen darstellst,
- wie du gestapelte Balken (`stack`) konfigurierst,
- wie du Overdrive-Balken für überlaufende Werte verstehst,
- wie du ChartStructure mit `filter` auf bestimmte Datensegmente einschränkst,
- wie du Attribute von übergeordneten Layouts erbst (`inherited-*`),
- und wie du eine ChartStructure als wiederverwendbare Component definierst.

---

01 Minimalkonfiguration

Ein ChartStructure braucht mindestens ein `dataset`:

```
- kind: ChartStructure
  spec:
    dataset: bb_example_dataset
```

BinoBI rendert dann ein Balkendiagramm mit den Kategorien aus dem DataSet. In der Praxis wirst du fast immer weitere Attribute setzen.

---

02 Attribute im Überblick

02.01 Datenanbindung, Hierarchie und Sortierung

Attribut	Typ	Standard	Beschreibung
`dataset`	string / array	–	Name des DataSets oder Array von DataSet-Namen (Pflichtfeld).
`chartTitle`	string	`""`	Optionaler Titel über dem Diagramm.
`filter`	string	`""`	SQL-`WHERE`-Fragment zur Einschränkung der Zeilen vor der Darstellung (z. B. `"region_group = 'EMEA'"`).
`level`	string	auto mit Warnung	Aggregationsebene. Werte: `auto`, `rowgroup`, `rowgroupindex`, `category`, `categoryindex`, `subcategory`, `subcategoryindex`. Die `…index`-Varianten behalten die Reihenfolge des DataSets bei. `auto` erkennt die Ebene automatisch aus dem DataSet (ohne Warnung). Bei leerem Feld erfolgt die Erkennung mit Warnung.
`order`	string	auto mit Warnung	Sortierschlüssel. Werte: `category`, `categoryindex`, `rowgroup`, `rowgroupindex`, `ac1` – `ac4`, `fc1` – `fc4`, `pp1` – `pp4`, `pl1` – `pl4`, `auto`, `inherited-closest`, `inherited-page`. Bei Szenario-Namen muss das Szenario in `scenarios` enthalten sein.
`orderDirection`	string	`asc`	Sortierrichtung: `asc`, `desc`, `inherited-closest`, `inherited-page`.
`limit`	integer	`0`	Maximale Balken pro Gruppe. Überschüssige Zeilen werden als **REST** zusammengefasst. `0` bedeutet unbegrenzt.

02.02 Szenarien und Varianzen

Attribut	Typ	Standard	Beschreibung
`scenarios`	string / array	–	Szenarien, die als Balkengruppen dargestellt werden (bis zu 4). Werte: `ac1` – `ac4`, `fc1` – `fc4`, `pp1` – `pp4`, `pl1` – `pl4`, `auto`, `inherited-closest`, `inherited-page`.

Attribut	Typ	Standard	Beschreibung
`variances`	string / array	–	Varianz-Definitionen nach dem Muster `d<szenarioB>_<szenarioA>_[pos|neg|neu]`. Das Schema ist in Kapitel 202, Abschnitt 03 erklärt. Auch `inherited-closest` und `inherited-page` möglich.

02.03 Maßeinheiten und Skalierung

Attribut	Typ	Standard	Beschreibung
`measureScale`	string	`GREATEST`	SI-Präfix für Datenskalierung. Werte: `_`, `k`, `M`, `G`, `T`, `P`, `E`, `Z`, `Y`, `m`, `µ`, `n`, `p`, `f`, `a`, `z`, `y`, `GREATEST`, `LEAST`.
`measureUnit`	string	`""`	Einheitenbezeichnung in der Achsenbeschriftung (z. B. `EUR`).
`unitScaling`	string/number	–	Name einer ScalingGroup oder numerischer Pixel-pro-Einheit-Faktor. Bei numerischem Wert werden Overdrive-Balken aktiviert (siehe Abschnitt 05).
`percentageScaling`	string/number	–	Name einer ScalingGroup oder numerischer Pixel-pro-Prozentpunkt-Faktor für Varianz-Nadeln.
`pixelPerUnit`	number	–	Explizite Pixel-pro-Einheit-Überschreibung (Mindestwert: 0). Übersteuert `unitScaling`.
`showMeasureScale`	bool	`false`	Zeigt den Skalierungspräfix in der Diagrammüberschrift an.

02.04 Darstellung

Attribut	Typ	Standard	Beschreibung
`showCategories`	bool	`true`	Zeigt Kategorielabels an den Balken an.
`scale`	string/number	auto mit Warnung	Proportionale Skalierung (Schriftgröße, Abstände, Balkenlängen). Skalierung greift bei Höhen-Überlauf (vertikale Labels

Attribut	Typ	Standard	Beschreibung
			abgeschnitten). Werte: leer = Auto mit `AUTO_scale`-Warnung; `"none"` = keine Skalierung; `"auto"` = Auto ohne Warnung; positive Zahl = fixer Faktor.

02.05 Internationalisierung

Attribut	Typ	Standard	Beschreibung
`i18nNamespace`	string	`_system`	i18n-Namensraum, in dem die Beschriftungen dieser Komponente aufgelöst werden. Fehlt das Feld, wird der Namensraum vom nächsten übergeordneten Element (Card, Seite, Tree, Grid oder Artefakt) geerbt; Schlüssel, die dort fehlen, fallen auf `_system` zurück.

Details zu Namensräumen und Übersetzungsdateien stehen in Kapitel 208.

---

03 Gestapelte Balken mit `stack`

Das optionale `stack`-Objekt aktiviert gestapelte Balken. Es kann mit `bar`-Modus verwendet werden.

```
spec:
  stack:
    by: scenarios # scenarios | dimensions
    mode: absolute # absolute | relative | absolute-relative
    order: dataset # asc | desc | dataset
```

Attribut	Typ	Standard	Beschreibung
`stack.by`	string	–	**Pflichtfeld.** `scenarios`: stapelt Szenarien hintereinander als Segmente. `dimensions`: leitet die Stapelspalte automatisch aus `level` ab (`rowGroup` → `category`, `category` → `subCategory`). Bei `dimensions` wird nur das erste Szenario verwendet; weitere werden mit Warnung verworfen.
`stack.mode`	string	`absolute`	Stapelmodus: `absolute` (additive Werte), `relative` (100 % gestapelt), `absolute-relative` (absolute

Attribut	Typ	Standard	Beschreibung
			Balkenhöhe mit Prozent-Labels).
`stack.order`	string	`dataset`	Reihenfolge der Stapelsegmente: `asc`, `desc` oder `dataset` (Reihenfolge aus dem DataSet).

03.01 Beispiel: Gestapelte Balken

```
# Absolut gestapelt – Szenarien AC und PP übereinandergelegt
- kind: ChartStructure
  spec:
    dataset: revenue_by_region
    chartTitle: "Revenue by region (gestapelt)"
    level: category
    order: ac1
    orderDirection: desc
    unitScaling: 0.003
    scenarios: [ac1, pp1]
    stack:
      by: scenarios
      mode: absolute
```

ChartStructure: Absolut gestapelt – Szenarien AC und PP übereinandergelegt

```
# Relativ gestapelt – Anteile der Szenarien in Prozent
- kind: ChartStructure
  spec:
    dataset: revenue_by_region
    chartTitle: "Revenue-Anteile"
    level: category
    order: category
    orderDirection: asc
    unitScaling: 150
    scenarios: [ac1, pp1, fc1]
    stack:
      by: scenarios
      mode: relative
```

ChartStructure: Relativ gestapelt – Anteile der Szenarien in Prozent

04 Vererbte Werte (Inherited Props)

Statt `scenarios`, `variances`, `order` und `orderDirection` auf jeder Komponente zu wiederholen, können diese vom übergeordneten `LayoutPage` oder `LayoutCard` geerbt werden:

Schlüsselwort	Quelle
`inherited-closest`	Nächster übergeordneter `LayoutCard` oder `LayoutPage`
`inherited-page`	Nächster übergeordneter `LayoutPage` (überspringt Cards)

```
# LayoutPage definiert gemeinsame Szenarien und Varianzen
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: revenue_page
spec:
  titleScenarios: [ac1, fc1]
  titleVariances: [dac1_fc1_pos]
  titleOrder: ac1
  titleOrderDirection: desc
  pageLayout: split-vertical
  children:
    - kind: ChartStructure
      spec:
        dataset: revenue_by_region
        level: category
        scenarios: inherited-page # übernimmt [ac1, fc1] von der Seite
        variances: inherited-page # übernimmt [dac1_fc1_pos]
        order: inherited-page # übernimmt ac1
        orderDirection: inherited-page # übernimmt desc
        unitScaling: 0.001
    - kind: Table
      spec:
        dataset: revenue_by_region
        scenarios: inherited-page
        variances: inherited-page
```

05 Overdrive-Balken bei Überlauf

Wenn `unitScaling` auf einen **numerischen Wert** (nicht `auto` oder ScalingGroup-Name) gesetzt ist und Balken die verfügbare Containerbreite überschreiten, greift das Overdrive-Rendering statt stiller Abschneidung:

1. **Basisabschnitt** – Der Balken wird bis zu einer berechneten Schnittbreite gezeichnet.
2. **Bruchanzeiger** – Eine vertikale Zickzack-Linie über die Balkenhöhe signalisiert den Achsenbruch.
3. **Overdrive-Abschnitt** – Eine proportional skalierte Verlängerung jenseits des Bruchs zeigt, wie weit der Balken den Schnitt übersteigt.

Balken, die innerhalb des Containers liegen, werden normal gerendert. Unter den überlaufenden Balken ist die Overdrive-Länge proportional zum Überlauf. AC-Szenario-Balken brechen an einer breiteren Stelle als PP-, FC- und PL-Balken – das erhält die IBCS-Hierarchie zwischen Szenariotypen.

> Overdrive gilt nur im Balkenmodus. Gestapelte Diagramme, relative Varianznadeldiagramme und auto-fit-Skalierung sind nicht betroffen.

---

06 Beispiel: ChartStructure als wiederverwendbare Component

In Kapitel 201 (Abschnitt 04.02) wurde diese Component definiert:

```
# components/203/203_example_chartstructure.yaml
apiVersion: bino.bi/v1alpha1
kind: ChartStructure
metadata:
  name: 202_country_chart
spec:
  dataset: bb_example_dataset_4_bn201
  level: category
  order: ac1
  orderDirection: desc
  showMeasureScale: true
  measureScale: k
  measureUnit: EUR
  limit: 3
```

```
  unitScaling: 0.0001
  scenarios: [ac1, pp1]
```

ChartStructure: Länder

06.01 Was hier konfiguriert ist

- **`level: category`** – Balken auf Category-Ebene (z. B. Länder innerhalb einer Region).
- **`order: ac1`** / **`orderDirection: desc`** – Sortiert nach Ist-Wert, größter Wert zuerst.
- **`limit: 3`** – Nur die drei größten Kategorien werden angezeigt; alle weiteren fließen in **REST**.
- **`scenarios: [ac1, pp1]`** – Zwei Szenarien: Ist (AC) und Vorperiode (PP) als nebeneinanderliegende Balken dargestellt (IBCS-konform: AC dunkelgrau, PP hellgrau).
- **`unitScaling: 0.0001`** – Pixel-pro-Einheit-Faktor, der in Kombination mit `measureScale: k` die gewünschte visuelle Balkenlänge ergibt.

---

07 Beispiel: ChartStructure mit Filter im Tree-Knoten

Aus dem Treiberanalyse-Baum (Kapitel 201, Abschnitt 04.03):

```
- id: category_1
  kind: ChartStructure
  ref: 202_country_chart
  optional: true
  spec:
    dataset: bb_example_dataset_4_bn201
    chartTitle: "Europe"
    filter: "rowGroup IN ('europe')"
```

Hier wird die Component `202_country_chart` als Basis verwendet. Die `spec`-Attribute überschreiben `dataset`, `chartTitle` und fügen einen `filter` hinzu. So zeigt dieser Knoten nur die europäischen Länder. Das Zusammenspiel von `ref` und `filter` ist in Kapitel 201, Abschnitt 04.05 erklärt.

---

08 ChartStructure vs. Table

Beide Komponenten zeigen strukturelle Daten (Kategorien, nicht Zeitreihen). Die Wahl hängt vom Kontext ab:

Kriterium	Table	ChartStructure
Darstellung	Tabellarisch mit Zahlen	Visuell als Balkendiagramm
Detail	Viele Spalten, exakte Werte	Wenige Szenarien, visueller Vergleich
Typischer Einsatz	Detail-Ansichten, Drill-Downs	Überblick, Treiberanalyse-Knoten
Gruppierung	Ja (`grouped: true`)	Nein (aber `stack` möglich)
Inline-Balken	Ja (`barColumns`)	Nicht nötig (ist selbst ein Diagramm)
Overdrive	Nein	Ja (bei numerischem `unitScaling`)
Inherited Props	Ja	Ja

In Tree-Komponenten werden ChartStructure häufig für übergeordnete Knoten und Table für Detail-Knoten verwendet (siehe Kapitel 201, Abschnitt 04).

---

09 Tipps

- **Limit bewusst setzen:** Bei vielen Kategorien werden die Balken sehr klein. Ein `limit` von 3–10 sorgt für ein lesbares Diagramm.
- **Sortierung:** `order: ac1` mit `orderDirection: desc` zeigt die Top-Performer zuerst – der häufigste Anwendungsfall.
- **Filter kombinieren:** Nutze `filter` in Kombination mit `ref` , um eine Basis-Component für verschiedene Datensegmente wiederzuverwenden.
- **`level: auto`** verwenden, wenn das DataSet eine klare Hierarchie hat – vermeidet Warnungen bei `bino lint` verglichen mit leerem `level` .
- **`showCategories: false`** bei sehr vielen Balken, wenn die Beschriftungen die Darstellung überladen.

> Die vollständige Attribut-Referenz für ChartStructure: cli.bino.bi/reference/chart-structure/ (https://cli.bino.bi/reference/chart-structure/)

204 – Die ChartTime-Komponente

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/chart-time/ (https://cli.bino.bi/reference/chart-time/).

Die `ChartTime`-Komponente stellt **Zeitreihen** als Balken- oder Liniendiagramm dar. Im Gegensatz zur `ChartStructure` (Kapitel 203), die Kategorien vergleicht, zeigt `ChartTime` die Entwicklung von Werten über die Zeit – z. B. monatliche Umsätze, quartalsweise Forecasts oder jährliche Trends.

In Kapitel 201 hast du ChartTime bereits in zwei Kontexten gesehen: als gestapeltes Zeitreihendiagramm auf Seite 1 (Abschnitt 03) und als Umsatzentwicklung auf der Drill-Down-Seite (Abschnitt 05). Dieses Kapitel erklärt die Komponente im Detail.

Du lernst:

- welche Attribute die ChartTime-Komponente steuern,
- wie du zwischen Balken-, Linien- und automatischem Modus wählst,
- wie du das Zeitintervall konfigurierst,
- wie du Durchschnitts- und Median-Overlays einblendest,
- wie du mit dem `stack`-Objekt gestapelte Diagramme und Flächendiagramme erzeugst,
- wie du Overdrive-Balken für überlaufende Werte verstehst,
- wie du Attribute von übergeordneten Layouts erbst (`inherited-*`),
- und wie du ChartTime mit Filtern und Parametern kombinierst.

---

01 Minimalkonfiguration

Ein ChartTime braucht mindestens ein `dataset`:

```
- kind: ChartTime
  spec:
    dataset: bb_example_dataset
```

BinoBI rendert dann ein Standard-Zeitreihendiagramm. In der Praxis wirst du fast immer `chartMode`, `dateInterval` und weitere Attribute setzen.

02 Attribute im Überblick

02.01 Darstellungsmodus und Zeitachse

Attribut	Typ	Standard	Beschreibung
`dataset`	string / array	–	Name des DataSets (Pflichtfeld).
`chartTitle`	string	`""`	Optionaler Titel über dem Diagramm.
`chartMode`	string	`bar`	Darstellungsart. Werte: `bar` (Balkendiagramm), `line` (Liniendiagramm), `auto` (wechselt zu `line`, wenn Datenpunkte `maxBars` überschreiten).
`maxBars`	integer	`28`	Schwellenwert für den automatischen Wechsel zu `line` bei `chartMode: auto`. Mindestwert: 1.
`lineFullWidth`	bool	`true`	Dehnt Liniendiagramme auf die volle Container-Breite aus. Hat keinen Effekt im Balkenmodus.
`axisLabelsMode`	string	–	Achsenbeschriftungs-Modus: `smart` (kontextabhängig), `long`, `short`.
`dateInterval`	string	auto-detect	Zeitgranularität. Werte: `year`, `quarter`, `month`, `week`, `day`, `hour`, `minute`, `second`, `millisecond`, `auto`. `auto` wählt das Intervall automatisch aus den DataSet-Metadaten.
`filter`	string	`""`	AlaSQL-`WHERE`-Fragment zur Einschränkung der Zeilen (z. B. `"rowGroup = '${REGION}'"`).

02.02 Aggregationsebene und Sortierung

Attribut	Typ	Standard	Beschreibung
`level`	string	auto mit Warnung	Aggregationsebene. Werte: `auto`, `rowgroup`, `rowgroupindex`, `category`, `categoryindex`, `subcategory`, `subcategoryindex`. Die `…index`-Varianten behalten die Reihenfolge des DataSets bei. `auto` erkennt die Ebene aus dem DataSet (ohne Warnung). Leer = Erkennung mit Warnung.

Attribut	Typ	Standard	Beschreibung
`order`	string	auto mit Warnung	Sortierschlüssel. Werte: `category`, `categoryindex`, `rowgroup`, `rowgroupindex`, `ac1` – `ac4`, `fc1` – `fc4`, `pp1` – `pp4`, `pl1` – `pl4`, `auto`, `inherited-closest`, `inherited-page`.
`orderDirection`	string	`asc`	Sortierrichtung: `asc`, `desc`, `inherited-closest`, `inherited-page`.
`limit`	integer	`0`	Stapel-Limit für Säulen. `0` bedeutet unbegrenzt.
`intervalSpanLimit`	integer	`28`	Steuert, wie viele Spalten die automatische Datumsintervall-Auswahl bestimmen.

02.03 Szenarien und Varianzen

Attribut	Typ	Standard	Beschreibung
`scenarios`	string / array	–	Szenarien (bis zu 4). Werte: `ac1` – `ac4`, `fc1` – `fc4`, `pp1` – `pp4`, `pl1` – `pl4`, `auto`, `inherited-closest`, `inherited-page`.
`variances`	string / array	–	Varianz-Definitionen nach dem Muster `d<szenarioB>_<szenarioA>_[pos|neg|neu]`. Leere Liste `[]` deaktiviert Varianzen. Auch `inherited-*` möglich.

02.04 Maßeinheiten und Skalierung

Attribut	Typ	Standard	Beschreibung
`measureScale`	string	`GREATEST`	SI-Präfix. Werte: `_`, `k`, `M`, `G`, `T`, `P`, `E`, `Z`, `Y`, `m`, `µ`, `n`, `p`, `f`, `a`, `z`, `y`, `GREATEST`, `LEAST`.
`measureUnit`	string	`""`	Einheitenbezeichnung an der Achse (z. B. `EUR`).
`unitScaling`	string/number	–	Name einer ScalingGroup oder numerischer Pixel-pro-Einheit-Faktor. Numerische Werte aktivieren Overdrive-Balken bei Überlauf (siehe Abschnitt 06).

Attribut	Typ	Standard	Beschreibung
`percentageScaling`	string/number	–	Name einer ScalingGroup oder numerischer Pixel-pro-Prozentpunkt-Faktor.
`syncSpaceLeft`	number	`-1`	Synchronisiert den linken Achsenabstand in Pixel über mehrere Diagramme. `-1` deaktiviert die Synchronisierung.

02.05 Sichtbarkeit und Overlays

Attribut	Typ	Standard	Beschreibung
`showCategories`	bool	`true`	Zeigt Kategorielabels an.
`showMeasureScale`	bool	`false`	Zeigt den Skalierungspräfix in der Diagrammüberschrift.
`showOverlayAvg`	bool	`true`	Zeigt eine horizontale Durchschnittslinie über alle Zeitpunkte.
`showOverlayMedian`	bool	`false`	Zeigt eine horizontale Medianlinie über alle Zeitpunkte.

02.06 Proportionale Skalierung

Attribut	Typ	Standard	Beschreibung
`scale`	string/number	auto mit Warnung	Proportionale Skalierung (Schriftgröße, Abstände, Säulendimensionen). Skalierung greift bei Breiten-Überlauf (horizontale Achsenlabels abgeschnitten). Werte: leer = Auto mit `AUTO_scale`-Warnung; `"none"` = keine; `"auto"` = Auto ohne Warnung; positive Zahl = fixer Faktor.

02.07 Internationalisierung

Attribut	Typ	Standard	Beschreibung
`i18nNamespace`	string	`_system`	i18n-Namensraum, in dem die Beschriftungen dieser Komponente aufgelöst werden. Fehlt das Feld, wird der Namensraum vom nächsten übergeordneten

Attribut	Typ	Standard	Beschreibung
			Element (Card, Seite, Tree, Grid oder Artefakt) geerbt; Schlüssel, die dort fehlen, fallen auf `_system` zurück.

Details zu Namensräumen und Übersetzungsdateien stehen in Kapitel 208.

---

03 Chart-Modi im Detail

03.01 Balkenmodus (`chartMode: bar`)

Standardmodus. Jeder Zeitpunkt wird als Gruppe von Balken dargestellt – ein Balken pro Szenario, IBCS-konform:

- **AC**: dunkelgraue Vollbalken
- **PP**: hellgraue Vollbalken
- **FC**: dunkelgraue Schraffur
- **PL**: dunkelgrau umrandet

03.02 Linienmodus (`chartMode: line`)

Geeignet für Trends und Vergleiche über viele Zeitpunkte. Besonderheiten gegenüber dem Balkenmodus:

- Quadratische Marker (8×8 px) an jedem Datenpunkt
- Datenwerte über jedem Marker angezeigt
- Szenario-Labels rechts neben dem letzten Datenpunkt
- Intelligente X-Achsen-Beschriftung: Start, Ende und Jahresgrenzen
- Vertikale gestrichelte Trennlinien an Jahresübergängen
- IBCS-Farbschema: AC = schwarz solid, PP = grau solid, FC = weiß / schwarz gestrichelt, PL = weiß / schwarz solid

03.03 Automatischer Modus (`chartMode: auto`)

Wechselt automatisch von Balken zu Linien, wenn die Datenpunkte den `maxBars`-Schwellenwert überschreiten. Nützlich für Diagramme, die sowohl kurze als auch lange Datenreihen anzeigen können.

```
- kind: ChartTime
  spec:
```

```
    dataset: revenue_weekly
    chartTitle: "Wöchentlicher Umsatz"
    chartMode: auto
    maxBars: 15 # ab >15 Datenpunkten → automatisch Linienmodus
    dateInterval: week
    measureScale: M
    measureUnit: "EUR"
    scenarios: [ac1, fc1]
```

---

04 Zeitintervall (`dateInterval`)

Das `dateInterval`-Attribut steuert die Aggregationsgranularität der Zeitachse:

Wert	Aggregation
`year`	Jahreswerte
`quarter`	Quartalswerte
`month`	Monatswerte
`week`	Wochenwerte
`day`	Tageswerte
`hour`	Stundenwerte
`minute`	Minutenwerte
`second`	Sekundenwerte
`millisecond`	Millisekunden
`auto`	Automatische Erkennung aus dem DataSet

> `intervalSpanLimit` beeinflusst bei `dateInterval: auto`, wie viele Datenspalten die Entscheidung über das Intervall steuern (Standard: 28).

---

05 Gestapelte Diagramme mit `stack`

Das optionale `stack`-Objekt aktiviert gestapelte Darstellungen. Bei `chartMode: bar` entstehen gestapelte Balken, bei `chartMode: line` gestapelte Flächendiagramme.

```
spec:
  stack:
    by: scenarios # scenarios | dimensions
    mode: absolute # absolute | relative | absolute-relative
    order: dataset # asc | desc | dataset
```

Attribut	Typ	Standard	Beschreibung
`stack.by`	string	–	**Pflichtfeld.** `scenarios`: stapelt Szenarien übereinander als Segmente. `dimensions`: leitet die Stapelspalte aus `level` ab (`rowGroup` → `category`, `category` → `subCategory`). Bei `dimensions` wird nur das erste Szenario genutzt; weitere werden mit Warnung verworfen.
`stack.mode`	string	`absolute`	Stapelmodus: `absolute` (additive Werte), `relative` (100 % normalisiert), `absolute-relative` (absolute Balkenhöhe mit Prozent-Labels).
`stack.order`	string	`dataset`	Reihenfolge der Stapelsegmente: `asc`, `desc` oder `dataset`.

05.01 Beispiel: Gestapeltes Balkendiagramm

Aus Kapitel 201, Abschnitt 03.02:

```
- kind: ChartTime
  spec:
    dataset: bb_example_dataset_4_bn201
    chartMode: bar
    dateInterval: year
    measureScale: k
    measureUnit: "EUR"
    percentageScaling: 1
    unitScaling: auto
    stack:
      by: scenarios
```

```
      mode: absolute
      order: dataset
```

ChartTime: Gestapeltes Balkendiagramm

Was hier passiert:

1. `chartMode: bar` – Balkendiagramm.
2. `dateInterval: year` – Zeitachse auf Jahresebene aggregiert.
3. `stack.by: scenarios` – Szenarien übereinandergestapelt; jeder Balken zeigt die Summe aller Szenarien, farblich unterteilt.
4. `stack.mode: absolute` – Absolute Werte. Bei `relative` wäre jeder Balken auf 100 % normalisiert.
5. `unitScaling: auto` – BinoBI berechnet den Skalierungsfaktor automatisch.

05.02 Beispiel: 100 % gestapelt

```
- kind: ChartTime
  spec:
    dataset: revenue_monthly
    chartTitle: "Revenue-Anteile nach Szenario"
    dateInterval: month
    unitScaling: 100
    scenarios: [ac1, pp1, fc1]
    stack:
      by: scenarios
      mode: relative
```

ChartTime: Gestapeltes Balkendiagramm 100%

05.03 Beispiel: Gestapeltes Flächendiagramm

Kombination aus `chartMode: line` und `stack` ergibt ein gestapeltes Flächendiagramm:

```
- kind: ChartTime
  spec:
    dataset: revenue_monthly
    chartTitle: "Revenue-Trend (gestapelte Fläche)"
    chartMode: line
    dateInterval: month
```

```
    unitScaling: 0.05
    scenarios: [ac1, pp1, fc1]
    stack:
      by: scenarios
      mode: absolute
```

ChartTime: GGestapeltes Flächendiagramm

05.04 Stapelmodi im Vergleich

Modus	Darstellung
`absolute`	Absolute Werte gestapelt. Balkenhöhe = Summe aller Segmente.
`relative`	Normalisiert auf 100 %. Zeigt Anteile statt absoluter Werte.
`absolute-relative`	Absolute Balkenhöhe mit zusätzlichen Prozent-Labels pro Segment.

---

06 Overdrive-Balken bei Überlauf

Wenn `unitScaling` auf einen **numerischen Wert** gesetzt ist und Balken die verfügbare Container-Höhe überschreiten, greift das Overdrive-Rendering:

1. **Basisabschnitt** – Der Balken wird bis zu einer berechneten Schnitthöhe gezeichnet.
2. **Bruchanzeiger** – Eine Zickzack-Linie über die Balkenbreite signalisiert den Achsenbruch.
3. **Overdrive-Abschnitt** – Eine proportional skalierte Verlängerung oberhalb des Bruchs zeigt, wie weit der Balken die Schnitthöhe übersteigt.

Balken innerhalb des Containers werden normal gerendert. Unter den überlaufenden Balken ist die Overdrive-Höhe proportional zum Überlauf. AC-Szenario-Balken brechen an einer höheren Stelle als PP-, FC- und PL-Balken – das erhält die IBCS-Hierarchie.

> Overdrive gilt nur im Balkenmodus und nicht für gestapelte Diagramme, relative Varianzen oder auto-fit-Skalierung.

---

07 Vererbte Werte (Inherited Props)

Statt `scenarios`, `variances`, `order` und `orderDirection` auf jeder Komponente zu wiederholen, können diese vom übergeordneten `LayoutPage` oder `LayoutCard` geerbt

werden:

Schlüsselwort	Quelle
`inherited-closest`	Nächster übergeordneter `LayoutCard` oder `LayoutPage`
`inherited-page`	Nächster übergeordneter `LayoutPage` (überspringt Cards)

```
children:
  - kind: ChartTime
    spec:
      dataset: revenue_trend
      dateInterval: month
      level: category
      scenarios: inherited-page # übernimmt titleScenarios der Seite
      variances: inherited-page # übernimmt titleVariances
      order: inherited-page
      orderDirection: inherited-page
```

---

08 Beispiel: ChartTime mit Filter auf der Drill-Down-Seite

Aus Kapitel 201, Abschnitt 05.03:

```
- kind: ChartTime
  spec:
    dataset: cities_bb_dataset
    measureScale: k
    measureUnit: EUR
    unitScaling: 0.01
    scenarios: [ac1, fc1, pl1]
    variances: []
    percentageScaling: 1
    filter: "rowGroup = '${REGION}'"
```

ChartTime: ChartTime mit Filter auf der Drill-Down-Seite (filter: rowGroup='German')

Das `filter`-Attribut akzeptiert SQL-Ausdrücke und wird zur Build-Zeit mit dem Parameter `${REGION}` aufgelöst (siehe Kapitel 201, Abschnitt 05.02). `variances: []` deaktiviert explizit

alle Varianz-Spalten. Beachte, dass `measureType`, `dataFormat*`, `type` und `categoryWidth` reine `Table`-Attribute sind (siehe Kapitel 202) – `ChartTime` kennt sie nicht.

---

09 ChartTime vs. ChartStructure

Kriterium	ChartTime	ChartStructure
X-Achse	Zeit (Datum)	Kategorien
Typische Frage	„Wie hat sich der Umsatz entwickelt?"	„Welche Region hat den höchsten Umsatz?"
Stacking	Ja (Balken und Flächen)	Ja (nur Balken)
Chart-Modi	`bar`, `line`, `auto`	Nur Balken
Zeitintervall	Ja (`dateInterval`)	Nein
Overlays	Durchschnitt, Median	Nein
Overdrive	Ja (bei numerischem `unitScaling`)	Ja (bei numerischem `unitScaling`)
Inherited Props	Ja	Ja

---

10 Tipps

- **`dateInterval: auto`** ist in den meisten Fällen ein guter Start – BinoBI wählt das passende Intervall aus den DataSet-Metadaten.
- **`chartMode: auto`** **mit** **`maxBars`** ist ideal für Diagramme, die sowohl kurze Jahres- als auch lange Monatsdatenreihen anzeigen sollen.
- **Stacking mit `dimensions`:** Statt Szenarien übereinander zu stapeln, kann mit `stack.by: dimensions` die Dimension gestapelt werden. So zeigt z. B. ein Gesamtumsatz-Balken die Aufschlüsselung nach Regionen pro Zeitpunkt.
- **`showOverlayAvg: false`** setzen, wenn der Durchschnitt die Darstellung verwirrt (z. B. bei stark saisonalen Daten).
- **`lineFullWidth: false`** kann für Liniendiagramme mit fester Spaltenbreite sinnvoll sein, wenn eine Synchronisierung mit danebenliegenden Balkendiagrammen gewünscht ist.

> Die vollständige Attribut-Referenz für ChartTime: cli.bino.bi/reference/chart-time/
> (https://cli.bino.bi/reference/chart-time/)

205 – Die Tree-Komponente

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/tree/ (https://cli.bino.bi/reference/tree/).

Die `Tree`-Komponente stellt **hierarchische Strukturen** als Baumdiagramm dar. Typische Anwendungsfälle sind Treiberanalysen, Wasserfallzerlegungen oder Organisationshierarchien. Jeder Knoten im Baum ist selbst eine Visualisierungskomponente – z. B. eine `ChartStructure`, `Table` oder `Label`.

Im Gegensatz zu `ChartStructure` und `ChartTime` ist Tree **struktur-gesteuert**, nicht rein datengesteuert: Das Layout wird durch Kanten zwischen Knoten-IDs definiert, der Inhalt jedes Knotens wird separat konfiguriert.

In Kapitel 201 (Abschnitt 04) hast du einen vollständigen Treiberanalyse-Baum gebaut. Dieses Kapitel erklärt die Tree-Komponente im Detail und geht auf alle Attribute ein.

Du lernst:

- wie du Knoten (`nodes`) und Kanten (`edges`) definierst,
- welche globalen Attribute Layout und Darstellung des Baums steuern,
- welche Knotenarten (`Label`, `Table`, `ChartStructure`, `ChartTime`, `Image`) verfügbar sind,
- wie du Kanten individuell gestaltest (Farbe, Breite, Strichelung),
- wie du `ref`, `filter` und `optional` kombinierst,
- wie DAG-Strukturen (gerichtete azyklische Graphen) aufgebaut werden,
- und welche Richtungen und Kantenstile verfügbar sind.

---

01 Aufbau eines Tree

Ein Tree besteht aus drei Teilen:

1. **Globale Attribute** – Richtung, Abstände, Kantenstil.
2. **Kanten (`edges`)** – Verbindungen zwischen Knoten.
3. **Knoten (`nodes`)** – Die eigentlichen Visualisierungen.

```
- kind: Tree
  spec:
    direction: ltr
    edgeStyle: orthogonal
    levelSpacing: 60
    nodeSpacing: 20
    showOperators: true
    edges:
      - { from: "root", to: "child_1", operator: "+" }
      - { from: "root", to: "child_2", operator: "+" }
    nodes:
      - id: root
        kind: ChartStructure
        spec:
          dataset: my_dataset
          chartTitle: Gesamt
      - id: child_1
        kind: Table
        spec:
          dataset: my_dataset
          type: sum
          sumTitle: Segment A
          filter: "category IN ('A')"
      - id: child_2
        kind: Table
        spec:
          dataset: my_dataset
          type: sum
          sumTitle: Segment B
          filter: "category IN ('B')"
```

Der Baum muss genau **einen Wurzelknoten** haben: einen Knoten, der nur als `from`-Wert, aber nie als `to`-Wert einer Kante auftaucht.

---

02 Globale Attribute

Attribut	Typ	Standard	Beschreibung
`direction`	string	`ltr`	Richtung des Baums: `ltr` (links → rechts), `rtl` (rechts → links), `ttb` (oben → unten), `btt` (unten → oben).
`edgeStyle`	string	`orthogonal`	Stil der Verbindungslinien: `straight` (gerade), `orthogonal` (rechtwinklig), `curved` (Bezier-Kurven).
`levelSpacing`	number	`60`	Abstand zwischen den Hierarchieebenen in Pixeln. Mindestwert: 0.
`nodeSpacing`	number	`20`	Abstand zwischen Geschwisterknoten auf derselben Ebene in Pixeln. Mindestwert: 0.
`showOperators`	bool	`true`	Mathematische Operatoren an den Kanten anzeigen.

02.01 Richtung wählen

Die häufigste Richtung ist `ltr` (links nach rechts) – der Wurzelknoten steht links, Detail-Knoten rechts. Für vertikale Bäume (z. B. Org-Charts) eignet sich `ttb` (oben nach unten). Bei ROI-Zerlegungen ist `ltr` standard.

---

03 Kanten (`edges`)

Jede Kante verbindet zwei Knoten über ihre `id` :

```
edges:
  - from: "root"
    to: "child_1"
    operator: "+"
    label: "" # optionales eigenes Label statt Operator-Symbol
    style:
      color: "#333333"
      width: 2
      dasharray: "5,3" # gestrichelt
```

03.01 Kanten-Attribute

Attribut	Typ	Standard	Beschreibung
`edges[].from`	string	–	Die `id` des Quellknotens (Pflichtfeld).
`edges[].to`	string	–	Die `id` des Zielknotens (Pflichtfeld).
`edges[].operator`	string	–	Operator-Symbol zwischen Geschwisterknoten. Werte: `+`, `−`, `*`, `/`, `x`, `÷`, `none`. Wird nur mit `showOperators: true` angezeigt.
`edges[].label`	string	–	Eigenes Label, das statt des Operator-Symbols angezeigt wird.
`edges[].style`	object	–	Individuelle Kantenstil-Konfiguration (siehe unten).

03.02 Kantenstil (`edges[].style`)

Attribut	Typ	Beschreibung
`edges[].style.color`	string	Kantenfarbe als CSS-Farbwert (z. B. `"#ff0000"` oder `"red"`).
`edges[].style.width`	number	Kantenbreite in Pixeln.
`edges[].style.dasharray`	string	SVG-Strichmuster (z. B. `"5,5"` für gleichmäßige Striche).

```
edges:
  - from: "root"
    to: "child_1"
    operator: "+"
    style:
      color: "#0066cc"
      width: 2
  - from: "root"
    to: "child_2"
    style:
      color: "#cc0000"
      width: 1
      dasharray: "4,4" # gestrichelte Linie
```

03.03 Operatoren

Der `operator` dokumentiert die mathematische Beziehung zwischen Eltern- und Kindknoten visuell:

Operator	Bedeutung
`+`	Summe (Teile addieren sich zum Ganzen)
`−`	Differenz
`*` / `x`	Multiplikation / IBCS-konformes `x`
`/` / `÷`	Division
`none`	Kein Symbol

---

04 Knoten (`nodes`)

Jeder Knoten definiert eine Visualisierungskomponente an einer Position im Baum:

```
nodes:
  - id: category_1
    kind: ChartStructure
    ref: 202_country_chart # Basis-Component aus components/
    spec:
      chartTitle: "Europe"
      filter: "rowGroup IN ('europe')"
```

Mit `ref` genügt es, nur die zu überschreibenden Felder anzugeben – Pflichtfelder wie `dataset` kommen aus der referenzierten Component.

04.01 Knoten-Attribute

Attribut	Typ	Beschreibung
`nodes[].id`	string	Eindeutiger Bezeichner – wird in `edges` als `from` / `to` referenziert. Pflichtfeld.
`nodes[].kind`	string	Komponentenart: `Label` , `Table` , `ChartStructure` , `ChartTime` , `Image` . Pflichtfeld.
`nodes[].ref`	string	Name einer wiederverwendbaren Component (standalone-Dokument). Nicht für `Label` unterstützt.

Attribut	Typ	Beschreibung
`nodes[].optional`	string	Wenn `true`, wird der Knoten ausgelassen, falls keine Daten vorhanden sind statt einen Fehler zu werfen
`nodes[].spec`	object	Komponentenspezifische Attribute. Überschreiben die Werte aus `ref`. Pflichtfeld wenn kein `ref` gesetzt.

---

05 Knotenarten (`kind`)

05.01 `Label`

Einfacher Text- oder HTML-Inhalt. Labels werden ausschließlich inline definiert – `ref` wird nicht unterstützt. Geeignet für KPI-Boxen, Kennzahlen oder kurze Annotationen.

```
- id: roi
  kind: Label
  spec:
    value: "<strong>ROI</strong><br/>12.5%"
    dataset: metrics_dataset # optional für Template-Interpolation
```

Das `value`-Feld akzeptiert HTML. Damit lassen sich einfache KPI-Boxen ohne vollwertige Komponenten erstellen:

```
- id: revenue_kpi
  kind: Label
  spec:
    value: |
      <div class='kpi-box'>
        <span class='label'>Revenue</span>
        <span class='value'>600M EUR</span>
      </div>
```

05.02 `Table`

Bettet eine vollwertige Table-Komponente (Kapitel 202) in den Baumknoten ein. Kann inline oder per `ref` definiert werden.

```
# Inline
- id: details
  kind: Table
  spec:
    dataset: revenue_breakdown
    scenarios: [ac1, py1]

# Per Referenz
- id: summary
  kind: Table
  ref: revenue_summary_table # Name in metadata.name des Table-Dokuments
  spec:
    limit: 5 # Überschreibt nur dieses eine Attribut
```

05.03 `ChartStructure`

Bettet ein strukturelles Balkendiagramm (Kapitel 203) in den Baumknoten ein.

```
# Inline
- id: breakdown
  kind: ChartStructure
  spec:
    dataset: cost_breakdown
    scenarios: [ac1]
    level: category

# Per Referenz mit Filter-Überschreibung
- id: revenue_chart
  kind: ChartStructure
  ref: base_revenue_chart
  spec:
    filter: "region = 'EMEA'" # überschreibt nur den Filter
```

05.04 `ChartTime`

Bettet ein Zeitreihendiagramm (Kapitel 204) in den Baumknoten ein.

```
- id: trend
  kind: ChartTime
  spec:
```

```
    dataset: monthly_revenue
    chartMode: line
    dateInterval: month
```

05.05 `Image`

Bettet ein Bild in den Baumknoten ein. Kann inline oder per `ref` (Asset-Manifest) definiert werden.

```
# Inline
- id: logo
  kind: Image
  spec:
    source: "./assets/company-logo.png"

# Per Referenz auf Asset-Manifest
- id: icon
  kind: Image
  ref: status_icon
```

---

06 DAG-Unterstützung

Der Tree unterstützt **gerichtete azyklische Graphen (DAGs)**, bei denen ein Knoten mehrere Eltern haben kann. Das ist nützlich, um gemeinsam genutzte Abhängigkeiten oder konvergierende Flüsse darzustellen.

```
edges:
  - { from: "a", to: "c" }
  - { from: "b", to: "c" } # Knoten "c" hat zwei Eltern
  - { from: "c", to: "d" }
nodes:
  - id: a
    kind: Label
    spec:
      value: "Input A"
  - id: b
    kind: Label
    spec:
```

```
      value: "Input B"
  - id: c
    kind: Label
    spec:
      value: "Shared Node"
  - id: d
    kind: Label
    spec:
      value: "Output"
```

---

07 Vollständiges Beispiel: ROI-Treiberanalyse

```
apiVersion: bino.bi/v1alpha1
kind: Tree
metadata:
  name: roi_driver_tree
  description: Return on Investment decomposition
spec:
  edges:
    - { from: "roi", to: "ros", operator: "x" }
    - { from: "roi", to: "capital-turnover" }
    - { from: "ros", to: "profit", operator: "/" }
    - { from: "ros", to: "revenue" }
    - { from: "capital-turnover", to: "revenue", operator: "/" }
    - { from: "capital-turnover", to: "capital" }
  direction: ltr
  edgeStyle: orthogonal
  levelSpacing: 80
  nodeSpacing: 30
  showOperators: true
  nodes:
    - id: roi
      kind: Label
      spec:
        value: "<div class='kpi-box'><span class='label'>ROI</span><span class='
    - id: ros
      kind: Label
      spec:
        value: "<div class='kpi-box'><span class='label'>ROS</span><span class='
```

```
    - id: capital-turnover
      kind: Label
      spec:
        value: "<div class='kpi-box'><span class='label'>Capital Turnover</span>
    - id: profit
      kind: Label
      spec:
        value: "<div class='kpi-box'><span class='label'>Profit</span><span clas
    - id: revenue
      kind: Label
      spec:
        value: "<div class='kpi-box'><span class='label'>Revenue</span><span cla
    - id: capital
      kind: Label
      spec:
        value: "<div class='kpi-box'><span class='label'>Capital</span><span cla
```

---

08 Vollständiges Beispiel: Gemischter Baum mit Charts und Tabellen

Das Treiberanalyse-Beispiel aus Kapitel 201 zeigt einen Baum mit drei Ebenen:

Ebene	Knoten-Art	DataSet	Inhalt
Wurzel	`ChartStructure`	`bb_example_dataset_4_bn201_regions`	Worldwide (alle Regionen)
Mitte	`ChartStructure`	`bb_example_dataset_4_bn201`	Pro Region (Europe, Asia, ...)
Blatt	`Table`	`bb_example_dataset`	Pro Land (Germany, France, ...)

Jede Ebene verwendet ein anderes DataSet mit einer anderen Hierarchie-Perspektive (siehe Kapitel 201, Abschnitt 02.04). Die Component-Vorlagen (`202_country_chart` , `201_city_table`) werden per `ref` eingebunden und durch `filter` auf den jeweiligen Datenausschnitt eingeschränkt.

Das vollständige Manifest findest du in Kapitel 201, Abschnitt 04.03.

---

09 Wiederverwendbare Components mit `ref` und Überschreibungen

```
# Standalone Table als Basis-Definition
---
apiVersion: bino.bi/v1alpha1
kind: Table
metadata:
  name: cost_breakdown_table
spec:
  dataset: cost_data
  scenarios: [ac1, py1]
  variances: [dpy1_ac1_neg]
---
# Tree, der die Component per ref einbindet
apiVersion: bino.bi/v1alpha1
kind: Tree
metadata:
  name: cost_tree
spec:
  edges:
    - { from: "total", to: "breakdown" }
  nodes:
    - id: total
      kind: Label
      spec:
        value: "Total Costs"
    - id: breakdown
      kind: Table
      ref: cost_breakdown_table
      spec:
        limit: 5 # überschreibt nur dieses Attribut, alles andere kommt aus dem
```

---

10 Tipps

- **`levelSpacing` und `nodeSpacing`** – Passe diese Werte an, wenn der Baum zu gedrängt oder zu weit auseinander wirkt. In der Vorschau (`bino preview`) siehst du das Ergebnis sofort.

- **Naming-Konvention für `id`** – Verwende sprechende Namen (z. B. `region_europe`, `country_germany`), um die `edges`-Definition lesbar zu halten.
- **Kantenstil für Hierarchien hervorheben** – Nutze `edges[].style.width` und `edges[].style.color`, um wichtige Kanten optisch von untergeordneten zu unterscheiden.
- **`edgeStyle: curved`** – Bezier-Kurven wirken organischer als rechtwinklige Linien; besonders geeignet für Org-Charts und konvergierende DAG-Strukturen.
- **DAG-Nutzung** – Ein Knoten darf mehrere Eltern haben. Stelle sicher, dass der Graph azyklisch bleibt (keine Zyklen), sonst führt bino lint einen Fehler aus.
- **Weniger ist mehr** – Ein Baum mit zu vielen Knoten wird unübersichtlich. Nutze `limit` in den Knoten-Komponenten, um die Darstellung kompakt zu halten.

> Die vollständige Attribut-Referenz für die Tree-Komponente: cli.bino.bi/reference/tree/ (https://cli.bino.bi/reference/tree/)

206 – Die Text-Komponente

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/text/ (https://cli.bino.bi/reference/text/).

Die `Text`-Komponente rendert **Markdown-Inhalte** direkt auf der Berichtsseite. Sie eignet sich für Erläuterungen, Kommentare, Legenden, KPI-Beschriftungen oder beliebige Textblöcke, die neben Tabellen und Diagrammen platziert werden sollen. Optional kann sie ein DataSet binden und darin enthaltene Werte per Template-Ausdruck direkt in den Text einbetten.

In Kapitel 201 (Abschnitt 03.03) hast du die Text-Komponente bereits als Teil eines Custom-Template-Layouts verwendet. Dieses Kapitel erklärt die Komponente im Detail.

Du lernst:

- welche Attribute die Text-Komponente steuern,
- welche Markdown-Elemente unterstützt werden,
- wie du dynamische Datenwerte per Template-Ausdruck einbettest,
- wie du Bilder aus Asset-Manifesten einbindest,
- wie die Skalierung funktioniert,
- und wie du Text in verschiedenen Layouts platzierst.

---

01 Minimalkonfiguration

Eine Text-Komponente braucht mindestens ein `value`-Feld mit Markdown-Inhalt:

```
- kind: Text
  spec:
    value: |
      ## Überschrift

      Dies ist ein Absatz mit **fettem** und *kursivem* Text.
```

Plain-Text ohne Markdown-Syntax ist ebenfalls gültig – er wird automatisch in einen `<p>`-Tag eingebettet.

---

02 Attribute im Überblick

Attribut	Typ	Standard	Beschreibung
`value`	string	–	Markdown-Text (Pflichtfeld). Verwende den YAML-Block-Stil (`|`) für mehrzeiligen Text. Unterstützt Markdown-Syntax und Template-Ausdrücke `${...}` .
`dataset`	string/array	–	Optionale Referenz auf ein oder mehrere DataSets für Template-Interpolation. Einzelner Name oder Array von Namen.
`scale`	string/number	auto mit Warnung	Steuert die Schriftgrößen-Skalierung (siehe Abschnitt 05).

Die Text-Komponente hat bewusst wenige Attribute – die Gestaltung erfolgt über Markdown und optionale Template-Ausdrücke.

---

03 Unterstützte Markdown-Elemente

Das `value` -Feld akzeptiert Standard-Markdown, der zur Build-Zeit in HTML konvertiert wird.

```
- kind: Text
  spec:
    value: |
      ## Überschrift Ebene 2

      ### Überschrift Ebene 3

      Normaler Text mit **Fettdruck** und *Kursivschrift*.
      _Du **kannst** sie kombinieren._

      ### Geordnete Liste

      1. Erster Punkt
      2. Zweiter Punkt
          1. Unterpunkt 2a
          2. Unterpunkt 2b

      ### Ungeordnete Liste
```

```
- Punkt A
- Punkt B

### Tabelle

| Region | Wachstum |
|--------|----------|
| EMEA   | +8%      |
| APAC   | +15%     |

### Blockquote

> Daten per Q4 2025

### Link

Quelle: [World Bank](https://data.worldbank.org)

### Bild aus Asset-Manifest

![Firmenlogo](asset:brandLogo)
```

Unterstützte Elemente:

Element	Markdown-Syntax
Überschriften	`##`, `###`, `####`
Fettdruck	`**text**`
Kursiv	`*text*` oder `_text_`
Kombiniert	`_Du **kannst** sie kombinieren_`
Geordnete Liste	`1.`, `2.`, …
Ungeordnete Liste	`-` oder `*`
Absätze	Leerzeile zwischen Textblöcken
Tabelle	`| Spalte | ... |`-Syntax
Blockquote	`> Text`

Element	Markdown-Syntax
Link	`[Linktext](URL)`
Bild (Asset)	`![Alt](asset:assetName)`

03.01 Asset-Bilder (`asset:`)

Das `asset:` -URI-Schema löst zur Build-Zeit auf die tatsächliche URL eines Asset-Manifests auf. So können Logos, Icons oder Grafiken direkt in Text-Komponenten eingebettet werden:

```
# Asset-Manifest definieren (assets/)
apiVersion: bino.bi/v1alpha1
kind: Asset
metadata:
  name: brandLogo
spec:
  source: "./resources/assets/logos/logo.png"
---
# In der Text-Komponente verwenden
- kind: Text
  spec:
    value: "![Firmenlogo](asset:brandLogo)"
```

---

04 Template-Ausdrücke

Das `value` -Feld unterstützt JavaScript-Template-Literal-Syntax (`${...}`) für dynamische Inhalte. Ausdrücke werden erst im Browser zur Laufzeit ausgewertet.

> **Wichtig: Dollarzeichen maskieren.** Bevor das Manifest die Render-Engine erreicht, ersetzt die CLI alle `${...}` -Vorkommen durch Umgebungsvariablen (siehe Kapitel 109). Ein unmaskiertes `${t('...')}` oder `${data...}` wird deshalb als Variablenname gelesen und der Build bricht ab:
>
> ```
> ✗ Configuration Error
>   unresolved environment variables: t('report.title') in pages/meine_seite.y
> ```
>
> Schreibe Template-Ausdrücke darum immer als `\${...}` . Der Backslash schützt den Ausdruck vor der Variablen-Ersetzung; die Engine bekommt ihn als `${...}` und wertet ihn

> aus. Weil YAML in doppelten Anführungszeichen kein `\$` erlaubt, verwende einen Block-Skalar (`>-` oder `|`) oder einfache Anführungszeichen.

04.01 Verfügbare Variablen

Variable	Beschreibung
`data.<datasetName>`	Array der gebundenen DataSet-Zeilen (Array von Objekten).
`data.<datasetName>[0].spalte`	Erster Wert der Spalte `spalte` aus dem DataSet.
`t('<schlüssel>')`	Gibt eine übersetzte Zeichenkette aus dem aktiven i18n-Manifest zurück.

Alle drei werden im `value` als `\${...}` geschrieben – siehe den Hinweis oben.

04.02 Beispiele

```
# Einzelnen Datenwert einbetten
- kind: Text
  spec:
    dataset: kpi_summary
    value: >-
      Umsatz: \${data.kpi_summary[0].ac1}
```

```
# Übersetzungsschlüssel verwenden
- kind: Text
  spec:
    value: >-
      \${t('report.title')}
```

```
# Kombinierter dynamischer Text
- kind: Text
  spec:
    dataset: kpi_summary
    value: |
      ### Quartalsübersicht

      **Umsatz:** \${data.kpi_summary[0].ac1} EUR
```

```
      **Plan:** ${data.kpi_summary[0].pl1} EUR
      **Abweichung:** ${data.kpi_summary[0].dac1_pl1_pos} EUR
```

```
# Mehrere DataSets binden
- kind: Text
  spec:
    dataset: [kpi_summary, region_data]
    value: |
      Region: ${data.region_data[0].category}
      Umsatz: ${data.kpi_summary[0].ac1}
```

04.03 Sicherheit

Template-Ausdrücke werden in einer **gesicherten Sandbox** ausgewertet – nur `data` und `t` sind zugänglich. Globale Browser-Objekte (`window` , `document` , `fetch` etc.) sind blockiert. Die HTML-Ausgabe wird gegen XSS bereinigt: Nur sichere Tags (`br` , `b` , `i` , `strong` , `em` , `span` , `p` , `div` , `table` , `a` etc.) und Attribute (`class` , `style` , `href` , `title` , `colspan` , `rowspan`) bleiben erhalten. Event-Handler (`onclick` , `onerror` …) und `javascript:` -URIs werden entfernt.

---

05 Skalierung (`scale`)

Das `scale` -Attribut steuert die Schriftgrößen-Skalierung der Komponente. Die Skalierung berücksichtigt sowohl Breite als auch Höhe – die engere Beschränkung gewinnt.

Wert	Verhalten
nicht gesetzt	Auto-Skalierung auf den verfügbaren Platz. Gibt eine `AUTO_scale` -Warnung mit dem angewendeten Faktor aus.
`"none"`	Keine Skalierung. Gibt eine `WARN_overflow` -Warnung aus, wenn der Inhalt überläuft.
`"auto"`	Auto-Skalierung, ohne Warnung.
`"0.5"` , `"0.8"` , `"1.2"` …	Wendet `baseFontSize × Faktor` an. Gibt `WARN_overflow` aus, wenn der Inhalt immer noch überläuft.

Ungültige Werte (negativ, nicht-numerisch, unbekannte Schlüsselwörter) fallen auf das Standardverhalten zurück und geben eine `ERR_scale` -Warnung aus.

```
# Auto-Skalierung ohne Warnung (empfohlen für Cards mit variabler Größe)
- kind: Text
  spec:
    value: "Dieser Text schrumpft, um in seinen Container zu passen."
    scale: auto

# Feste 80 %-Größe
- kind: Text
  spec:
    value: "Feste Schriftgröße bei 80 %."
    scale: "0.8"

# Keine Skalierung – für Seiten mit bekanntem Layout
- kind: Text
  spec:
    value: "Immer in Basis-Schriftgröße."
    scale: none
```

---

06 Beispiel aus dem Beispielbericht

Aus Kapitel 201, Abschnitt 03.02:

```
- kind: Text
  spec:
    scale: none
    value: |
      ## Important Text

      Ipsum dolor sit amet, consectetur adipiscing elit. Sed do eiusmod
      tempor incididunt ut labore et dolore magna aliqua.

      ### Markdown Elements Guide

      *This text will be italic*   **This text will be bold**
      _You **can** combine them_

       ### Ordered
```

```
      1. Item 1
      2. Item 2
          1. Item 2a
          2. Item 2b

      Ipsum dolor sit amet, consectetur adipiscing elit.

    dataset: bb_example_dataset_4_bn201
```

Das `dataset`-Attribut verknüpft die Komponente mit einem DataSet. Auch wenn im YAML oben keine `${...}`-Ausdrücke verwendet werden, ist die Verknüpfung vorhanden, um bei Bedarf Datenwerte dynamisch einzubetten.

---

07 Standalone-Manifest

Text kann auch als eigenständiges Dokument definiert und per `ref` eingebunden werden:

```
# texts/main_title.yaml
apiVersion: bino.bi/v1alpha1
kind: Text
metadata:
  name: mainTitle
spec:
  value: "Sales Overview Report"
```

```
# In einer LayoutPage per ref verwenden
children:
  - kind: Text
    ref: mainTitle
```

---

08 Text im Layout platzieren

Die Text-Komponente wird wie jede andere Komponente als Kind-Element in einer `LayoutPage` oder `LayoutCard` platziert. In Kombination mit `custom-template` (Kapitel 104, Abschnitt 06.03) kannst du Text und Diagramme frei anordnen:

```
pageLayout: custom-template
pageCustomTemplate: |
  "A A A A A B B"
  "A A A A A C C"
children:
  - kind: Table
    spec: ... # → Zelle A
  - kind: Text
    spec: ... # → Zelle B
  - kind: ChartTime
    spec: ... # → Zelle C
```

Die Zuordnung erfolgt alphabetisch: erstes Kind → A, zweites → B, drittes → C (siehe Kapitel 104, Abschnitt 06.03).

```
# Text als Layout-Kind (Inline-Definition)
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: sales_layout
spec:
  pageLayout: full
  children:
    - kind: Text
      spec:
        value: "Sales Overview Report"
        scale: auto
```

---

09 Tipps

- **YAML-Block-Stil (`|`):** Verwende immer `value: |` für mehrzeiligen Markdown. So bleiben Zeilenumbrüche und Einrückungen erhalten.
- `scale: auto` empfiehlt sich für Cards mit variabler Breite; `scale: none` für Seiten mit bekanntem fixen Layout.
- **Template-Ausdrücke für dynamische Berichte:** Binde ein DataSet und nutze `${data.<name>[0].feld}`, um Schlüsselkennzahlen direkt im Text zu nennen – das erhöht die Aussagekraft neben Diagrammen.

- `asset:`**-Bilder** erlauben es, Logos und Icons versioniert über Asset-Manifeste zu verwalten, statt Pfade direkt im Text hardzucodieren.
- **Markdown-Tabellen** eignen sich für kompakte Vergleiche (z. B. Regionen × Wachstum), ohne eine vollwertige Table-Komponente zu benötigen.
- **Kein unsicheres HTML nötig:** Event-Handler und externe Scripts werden bereinigt. Nutze CSS-Klassen über Markdown-Attribute oder ComponentStyle für visuelle Anpassungen.

> Die vollständige Attribut-Referenz für die Text-Komponente: cli.bino.bi/reference/text/ (https://cli.bino.bi/reference/text/)

207 - ComponentStyle

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).

`ComponentStyle` bündelt Typografie, Farben und Feinabstimmungs-Werte (Scale-Faktoren, Padding) für die BinoBI-Komponenten an einer zentralen Stelle, statt sie in jeder Table/Chart/Tree-Spec einzeln zu wiederholen. Das Projekt bringt bereits eine Beispiel-Datei mit: `styles/corporate_design_extern.yaml`. Dieses Kapitel erklärt ihre Struktur im Detail.

Du lernst:

- welche Stil-Bereiche (`global`, `bn-text`, `bn-chart-time`, `bn-chart-structure`, `bn-tree`, `bn-grid`) es gibt,
- welche Attribute innerhalb dieser Bereiche zur Verfügung stehen,
- wie IBCS-Szenario- und Varianz-Farben pro Komponente überschrieben werden,
- wie das verwandte `ScalingGroup`-Manifest Skalierungen über mehrere Komponenten synchronisiert,
- und was du beim heutigen Stand von `ComponentStyle` beachten solltest.

---

01 Grundstruktur

```
apiVersion: bino.bi/v1alpha1
kind: ComponentStyle
metadata:
  name: corporateTheme
spec:
  content: {} # Objekt oder JSON-String
```

`spec.content` ist entweder ein JSON-String oder ein Objekt mit den folgenden Top-Level-Bereichen:

Bereich	Betrifft
`global`	Schriftart, Schriftgröße und Farben, projektweit.

Bereich	Betrifft
`bn-text`	Die `Text`-Komponente (aktuell wenige Attribute).
`bn-chart-time`	`ChartTime` (Kapitel 204).
`bn-chart-structure`	`ChartStructure` (Kapitel 203).
`bn-tree`	`Tree` (Kapitel 205).
`bn-grid`	`Grid` (Kapitel 107).

---

02 `global`: Typografie und Grundfarben

```
spec:
  content:
    global:
      fontSizePx: 13
      fontFamily: "'Arial', sans-serif"
      fontColor: "#333333"
      fontBackgroundColor: "#ffffff"
```

Attribut	Typ	Standard	Beschreibung
`fontSizePx`	number	`13.3333`	Basis-Schriftgröße in Pixel. Viele Scale-Faktoren (Abschnitt 03) sind Vielfache dieses Werts.
`fontFamily`	string	`'Noto Sans', sans-serif`	CSS `font-family`.
`fontColor`	string	`#000000`	Haupt-Textfarbe.
`fontBackgroundColor`	string	`#ffffff`	Hintergrundfarbe für Text-Elemente.

---

03 `bn-chart-time` / `bn-chart-structure`: Diagramm-Feinsteuerung

Beide Bereiche sind strukturell identisch aufgebaut:

```
spec:
  content:
    bn-chart-structure:
      scaleFactors:
        barWidth: 1.5
        categoryWidth: 2.0
      axisStyles:
        ac: { stroke: "#000000", fill: "#000000", width: 0.1 }
        fc: { stroke: "#000000", fill: "#ffffff", width: 0.1, dasharray: "10 5 5
        pp: { stroke: "#666666", fill: "#666666", width: 0.1 }
        pl: { stroke: "#000000", fill: "#ffffff", width: 0.1 }
      barStyles:
        ac: { stroke: "#00477e", fill: "#00477e", width: 0.1 }
        pp: { stroke: "#999999", fill: "#999999", width: 0.1 }
      varianceStyles:
        pos: { stroke: "#008f96", fill: "#008f96", width: 0.1 }
        neg: { stroke: "#c0392b", fill: "#c0392b", width: 0.1 }
        neu: { stroke: "#808080", fill: "#808080", width: 0.1 }
      stackColors: ["#000000", "#c0c0c0", "#808080"]
      padding: { top: 12, left: 12, right: 12, bottom: 12 }
```

Attribut	Beschreibung
`scaleFactors`	Dutzende Pixel-Multiplikatoren (z. B. `barWidth`, `columnWidth`, `categoryWidth`, `axisLineWidth`, `markerWidth`), jeweils multipliziert mit `global.fontSizePx`.
`axisStyles`	SVG-Stil (`stroke`, `fill`, `width`, `dasharray`, `fillpattern`) je IBCS-Szenario: `ac`, `fc`, `pp`, `pl`.
`barStyles`	Wie `axisStyles`, aber für die Balken selbst.
`varianceStyles`	SVG-Stil je Varianz-Kategorie: `pos`, `neg`, `neu`.
`stackColors`	Farbliste, durch die gestapelte Segmente (Kapitel 203/204, Abschnitt `stack`) der Reihe nach zyklisch durchlaufen.
`padding`	`top` / `left` / `right` / `bottom` in Pixel (Standard je `10`).

`bn-chart-time` kennt zusätzlich keine eigenen Extra-Attribute gegenüber `bn-chart-structure` – beide teilen sich dieselbe Struktur, lediglich die Namen der `scaleFactors`-Schlüssel unterscheiden sich leicht (z. B. `columnWidth` bei `bn-chart-time` statt `barWidth` bei `bn-chart-structure`), passend zur unterschiedlichen Diagrammausrichtung.

04 `bn-tree` und `bn-grid`

```
spec:
  content:
    bn-tree:
      edgeColor: "#00477e"
      edgeWidth: 1.5
      operatorCircleRadius: 12
      operatorFontSizeScale: 0.9
      operatorColor: "#00477e"
    bn-grid:
      headerBackground: "#f5f5f5"
      borderColor: "#cccccc"
      borderWidth: 1
      cellPadding: 8
```

`bn-tree` steuert die Kantenfarbe/-breite und das Aussehen der Operator-Kreise aus Kapitel 205. `bn-grid` steuert Kopfhintergrund, Rahmenfarbe/-breite und Zellabstand des `Grid` aus Kapitel 107.

---

05 Vollständiges Beispiel

Das Projekt enthält bereits eine Beispieldatei, die alle Bereiche kombiniert:

```
# styles/corporate_design_extern.yaml
apiVersion: bino.bi/v1alpha1
kind: ComponentStyle
metadata:
  name: corporate_design_extern
  description: Extern Styles
spec:
  content:
    global:
      fontSizePx: 13
      fontFamily: "'Arial', sans-serif"
      fontColor: "#333333"
      fontBackgroundColor: "#ffffff"
    bn-chart-structure:
```

```
      barStyles:
        ac: { stroke: "#00477e", fill: "#00477e", width: 0.1 }
        pp: { stroke: "#999999", fill: "#999999", width: 0.1 }
      varianceStyles:
        pos: { stroke: "#008f96", fill: "#008f96", width: 0.1 }
        neg: { stroke: "#c0392b", fill: "#c0392b", width: 0.1 }
      padding: { top: 12, left: 12, right: 12, bottom: 12 }
    bn-chart-time:
      barStyles:
        ac: { stroke: "#00477e", fill: "#00477e", width: 0.1 }
        pp: { stroke: "#999999", fill: "#999999", width: 0.1 }
      varianceStyles:
        pos: { stroke: "#008f96", fill: "#008f96", width: 0.1 }
        neg: { stroke: "#c0392b", fill: "#c0392b", width: 0.1 }
    bn-tree:
      edgeColor: "#00477e"
      operatorColor: "#00477e"
    bn-grid:
      headerBackground: "#f5f5f5"
      borderColor: "#cccccc"
      borderWidth: 1
      cellPadding: 8
```

`bino lint --execute-queries` validiert diese Datei vollständig gegen das Schema – alle Schlüssel unter `content.*` sind zusätzlich mit `additionalProperties: true` versehen, sodass auch neue, noch nicht in diesem Kapitel dokumentierte Unter-Attribute keinen Validierungsfehler auslösen.

---

06 Verwandt: `ScalingGroup` – Skalierung über Komponenten synchronisieren

Neben Farben und Typografie gehört zur visuellen Konsistenz auch, dass **Balkenlängen vergleichbar bleiben**. Genau dafür gibt es das kleine `ScalingGroup`-Manifest: ein benannter Skalierungsfaktor, den mehrere Charts und Tabellen gemeinsam referenzieren.

```
apiVersion: bino.bi/v1alpha1
kind: ScalingGroup
metadata:
  name: bn107_chart_scale
```

```
spec:
  value: 0.005
```

Attribut	Pflicht	Beschreibung
`value`	ja	Skalierungsfaktor (Zahl > 0), der für alle referenzierenden Komponenten gilt.

Referenziert wird die Gruppe überall dort, wo `unitScaling` (Pixel pro Dateneinheit) oder `percentageScaling` (Pixel pro Prozentpunkt) statt einer Zahl einen **Namen** akzeptieren – also auf `Table`, `ChartStructure` und `ChartTime` (siehe die Attribut-Tabellen in Kapitel 202-204):

```
- kind: ChartStructure
  spec:
    dataset: bb_example_dataset
    unitScaling: bn107_chart_scale # Name statt Zahl
```

Alle Komponenten mit derselben Gruppe verwenden garantiert denselben Pixel-pro-Einheit-Maßstab – auch über mehrere Seiten hinweg. Das Grid-Beispiel aus Kapitel 107 (`components/107/107_example_grid.yaml`) nutzt genau dieses Muster: Beide Chart-Zellen referenzieren `bn107_chart_scale`, sodass gleiche Werte in beiden Zellen gleich lange Balken ergeben. Nebeneffekt eines festen gemeinsamen Maßstabs: Überläuft ein Balken seinen Container, greift das Overdrive-Rendering (Kapitel 203, Abschnitt 06), statt dass die Komponente still ihren eigenen Maßstab wählt.

---

07 Was dieses Kapitel (noch) nicht beantworten kann

Weder das JSON-Schema noch die Referenz unter `cli.bino.bi` dokumentieren aktuell ein Feld, mit dem eine `LayoutPage`, `LayoutCard` oder ein `ReportArtefact` **gezielt** eine bestimmte `ComponentStyle`-Instanz auswählt (kein `styleRef`, `selectedStyle` o. Ä. ist Teil des aktuellen Schemas). Der offizielle Styling-Guide bestätigt das ausdrücklich: *"the exact wiring between styles/i18n and components depends on the front-end implementation"*. In diesem Projekt liegt entsprechend genau eine `ComponentStyle`-Datei vor. Vermutlich ist – analog zur Beschreibung in Kapitel 105 – vorgesehen, mehrere `ComponentStyle`-Varianten über `constraints` (z. B. `spec.format==a4` vs. `spec.format==xga` oder `labels.<key>`) kontextabhängig ein- oder auszublenden, so wie es beim DataSource-Sampling-Muster üblich ist.

> **Prüfe das für deine bino-Version:** Bevor du dich auf ein bestimmtes Aktivierungsverhalten verlässt, teste es mit deiner installierten `bino` - und Template-Engine-Version in `bino preview`, und ziehe die aktuelle Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/) zurate.

---

08 Zusammenfassung

1. **`ComponentStyle`** bündelt Typografie- und Farbwerte in `spec.content`, unterteilt in `global`, `bn-text`, `bn-chart-time`, `bn-chart-structure`, `bn-tree` und `bn-grid`.
2. **`global`** setzt Basis-Schriftgröße, -familie und -farbe für das ganze Projekt.
3. **`bn-chart-time` / `bn-chart-structure`** steuern `scaleFactors`, `axisStyles` / `barStyles` / `varianceStyles` je IBCS-Szenario (`ac` / `fc` / `pp` / `pl`) bzw. Varianz (`pos` / `neg` / `neu`), `stackColors` und `padding`.
4. **`bn-tree` / `bn-grid`** steuern Kanten- bzw. Rahmen-/Kopfzeilen-Optik der jeweiligen Komponente.
5. Der genaue Aktivierungsmechanismus für mehrere parallele `ComponentStyle`-Varianten ist derzeit nicht offiziell dokumentiert – teste ihn projektspezifisch.

> **Jetzt bist du dran!** Ändere in `styles/corporate_design_extern.yaml` die Farbe `bn-chart-structure.barStyles.ac` und beobachte in `bino preview`, ob und wie sich ein `ChartStructure` in deinem Projekt verändert.

208 - Internationalisierung

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).

Einzelne i18n-Bausteine sind dir bereits begegnet: `i18nNamespace` in Kapitel 104 und der Hinweis auf `t('key')` in Text-Templates in Kapitel 206. Dieses Kapitel führt das `Internationalization`-Manifest selbst ein und ordnet die verstreuten i18n-Felder ein.

Du lernst:

- wie ein `Internationalization`-Manifest aufgebaut ist (`code`, `namespace`, `content`),
- wie `i18nNamespace` eine Komponente, eine Seite oder ein ganzes Artefakt mit einem Namensraum verknüpft,
- wie du Übersetzungen mit `\${t('...')}` in einer `Text`-Komponente ausgibst,
- und wie du ein Artefakt pro Sprache baust.

---

01 Das Internationalization-Manifest

```
apiVersion: bino.bi/v1alpha1
kind: Internationalization
metadata:
  name: 208_example_products_de
spec:
  code: de
  namespace: products
  content:
    product.title: "Produktübersicht"
    product.subtitle: "Umsatz nach Region"
```

Attribut	Typ	Pflicht	Beschreibung
`code`	string	ja	Sprachcode, z. B. `de` oder `en`. Er muss **exakt** dem `spec.language` des Artefakts entsprechen – der Abgleich ist ein reiner Zeichenketten-Vergleich ohne

Attribut	Typ	Pflicht	Beschreibung
			BCP-47-Normalisierung, `de-DE` passt also nie zu `language: de`.
`namespace`	string	nein	Optionaler Namensraum, um Übersetzungsblöcke thematisch zu trennen (z. B. `products` statt des Standard-Namensraums `_system`).
`content`	object/string	ja	Schlüssel-Wert-Paare der übersetzten Texte, als YAML-Objekt oder JSON-String.

Für mehrere Sprachen legst du pro Sprache ein eigenes Dokument an – meist in derselben Datei, per `---` getrennt:

```
apiVersion: bino.bi/v1alpha1
kind: Internationalization
metadata:
  name: 208_example_products_de
spec:
  code: de
  namespace: products
  content:
    product.title: "Produktübersicht"
---
apiVersion: bino.bi/v1alpha1
kind: Internationalization
metadata:
  name: 208_example_products_en
spec:
  code: en
  namespace: products
  content:
    product.title: "Product Overview"
```

---

02 `i18nNamespace`: einen Namensraum verknüpfen

`i18nNamespace` legt fest, in welchem Namensraum die Beschriftungen einer Komponente aufgelöst werden. Das Feld gibt es auf dem `ReportArtefact`, auf `LayoutPage` und

`LayoutCard`, auf `Tree` und `Grid` sowie auf den Komponenten selbst (`Table`, `Text`, `ChartStructure`, `ChartTime`, …):

```
spec:
  i18nNamespace: products
```

Du musst es nicht überall wiederholen: fehlt das Feld, erbt die Komponente den Namensraum vom nächsten übergeordneten Element. Ein Schlüssel, der im gewählten Namensraum fehlt, fällt auf den Standard-Namensraum `_system` zurück. Das Projekt liefert dafür bereits ein `i18n/default.yaml` mit Schlüsseln wie `report.date` und `report.title` mit.

> **`titleNamespace` ist veraltet.** `LayoutPage` und `LayoutCard` kennen zusätzlich noch `titleNamespace`. Es wirkt nur auf den Titel und wird **nicht** an die Kinder vererbt. `bino lint` meldet es als `i18n-title-namespace-deprecated`. Verwende stattdessen `i18nNamespace`.

---

03 Übersetzungen ausgeben

Übersetzte Zeichenketten holst du dir in einer `Text`-Komponente mit `t('<schlüssel>')`:

```
- kind: Text
  spec:
    i18nNamespace: products
    value: >-
      \${t('product.title')}
```

> **Der Backslash gehört dazu.** Die CLI ersetzt `${...}` vorher durch Umgebungsvariablen. Ohne Maskierung bricht der Build mit `unresolved environment variables: t('product.title')` ab. Kapitel 206, Abschnitt 04 erklärt das im Detail.

Die übrigen Komponenten (`Table`, `ChartStructure`, `ChartTime`, …) haben **kein** eigenes Feld für einen Übersetzungsschlüssel. Sie kennen nur `i18nNamespace` und beschriften sich daraus selbst – Szenario-Labels wie `global.ac1` etwa kommen aus dem Katalog, nicht aus einem Attribut am Manifest.

---

04 Beispiel: dasselbe Artefakt in zwei Sprachen

Ein Katalog je Sprache, gleicher `namespace`, unterschiedlicher `code`:

```
apiVersion: bino.bi/v1alpha1
kind: Internationalization
metadata:
  name: demo_de
spec:
  code: de
  namespace: demo
  content:
    demo.hello: "Hallo Welt"
---
apiVersion: bino.bi/v1alpha1
kind: Internationalization
metadata:
  name: demo_en
spec:
  code: en
  namespace: demo
  content:
    demo.hello: "Hello World"
```

Eine Seite, die den Namensraum setzt und den Schlüssel ausgibt:

```
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: p_i18n
spec:
  pageFormat: a4
  pageOrientation: landscape
  i18nNamespace: demo
  children:
    - kind: Text
      spec:
        value: >-
          \${t('demo.hello')}
```

Und zwei Artefakte, die sich nur in `language` und `filename` unterscheiden:

```
apiVersion: bino.bi/v1alpha1
kind: ReportArtefact
metadata:
  name: r_de
spec:
  title: "Bericht"
  filename: "bericht_de.pdf"
  format: a4
  orientation: landscape
  language: de
  layoutPages:
    - page: p_i18n
---
apiVersion: bino.bi/v1alpha1
kind: ReportArtefact
metadata:
  name: r_en
spec:
  title: "Report"
  filename: "bericht_en.pdf"
  format: a4
  orientation: landscape
  language: en
  layoutPages:
    - page: p_i18n
```

`bino build` erzeugt daraus zwei PDFs: eines mit *Hallo Welt*, eines mit *Hello World*.

> **Achte auf das Format.** Ein Artefakt ohne `format` fällt auf `xga` zurück. Passt das nicht zum `pageFormat` der Seite, filtert bino die Seite weg und `bino lint` meldet `artefact-layoutpage-required` – das Artefakt wird gebaut, bleibt aber leer.

---

05 Typische Stolpersteine

- **`code` passt nicht zu `language`.** Der Vergleich ist exakt. Ein Katalog mit `code: en`, für den es kein Artefakt mit `language: en` gibt, wird nie verwendet; `bino lint` meldet das als `i18n-code-unused`.
- **`${...}` nicht maskiert.** Führt zu `unresolved environment variables` und bricht den Build ab (siehe Abschnitt 03).

- **`\$` in doppelten Anführungszeichen.** YAML erlaubt diese Escape-Sequenz dort nicht (`found unknown escape character`). Nimm einen Block-Skalar `>-` oder `|`.
- **`titleNamespace` statt `i18nNamespace`.** Wirkt nur auf den Titel und vererbt sich nicht.

---

06 Zusammenfassung

1. **`Internationalization`** definiert `code` (Sprache), optional `namespace` und `content` (Schlüssel-Wert-Paare).
2. Mehrere Sprachen sind mehrere Dokumente mit demselben `namespace`, aber unterschiedlichem `code`.
3. **`i18nNamespace`** verknüpft Artefakt, Seite, Card oder Komponente mit einem Namensraum und vererbt sich nach unten; fehlende Schlüssel fallen auf `_system` zurück. `titleNamespace` ist veraltet.
4. **`t('<schlüssel>')`** gibt einen übersetzten Text in einer `Text`-Komponente aus – immer als `\${t('...')}` maskiert.
5. Ein Artefakt pro Sprache: gleiche `LayoutPage`, unterschiedliches `language`.

> **Jetzt bist du dran!** Lege zwei `Internationalization`-Manifeste mit demselben Namensraum und den Codes `de` und `en` an, setze `i18nNamespace` auf einer `LayoutPage`, gib den Schlüssel per `\${t('...')}` in einer `Text`-Komponente aus und baue zwei Artefakte, die sich nur in `language` unterscheiden.

209 - LiveReportArtefact im Detail

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).

Kapitel 201 hat `LiveReportArtefact` bereits kurz eingeführt: eine Route `/`, die ein ganzes `ReportArtefact` zeigt, und eine Route `/drill`, die eine parametrisierte `LayoutPage` mit einem statischen `select`-Query-Parameter kombiniert. Dieses Kapitel vertieft `LiveReportArtefact`: mehrere Routen, alle Query-Parameter-Typen und dataset-gestützte Auswahllisten.

Du lernst:

- wie `routes` aufgebaut ist und was die Pflicht-Route `/` bedeutet,
- welche `queryParams`-Typen es gibt,
- wie du Auswahllisten statisch **oder** aus einem DataSet befüllst,
- und wie du das Ganze mit `bino serve --live` startest.

---

01 Grundstruktur

```
apiVersion: bino.bi/v1alpha1
kind: LiveReportArtefact
metadata:
  name: bn201_live
spec:
  title: BN201 Dashboard
  routes:
    /: # Pflicht-Route, dient als Startseite
      artefact: handbuch_examples_bn201
```

Attribut	Beschreibung
`title`	**Pflichtfeld.** Anzeigename der Live-Anwendung.
`routes`	**Pflichtfeld.** Map von URL-Pfad zu Routen-Konfiguration. Die Route `/` ist zwingend erforderlich.

Jede Route setzt **genau eines** von `artefact` (Name eines `ReportArtefact`) oder `layoutPages` (eine oder mehrere `LayoutPage`s) – nie beides gleichzeitig.

---

02 `queryParams`: Typen im Überblick

Typ	Beschreibung
`string`	Freitext (Standard, wenn `type` weggelassen wird).
`number`	Einzelner Zahlenwert.
`number_range`	Zahlenbereich (nutzt `options.min` / `max` / `step`).
`select`	Auswahlliste – entweder statisch (`options.items`) oder dataset-gestützt (`options.dataset`).
`date`	Einzelnes Datum.
`date_time`	Datum mit Uhrzeit.

Gemeinsame Felder aller Typen:

Attribut	Beschreibung
`name`	Name des Parameters (gültiger Bezeichner), im Manifest als `${NAME}` verwendbar.
`default`	Standardwert, falls der Parameter in der Anfrage fehlt.
`optional`	Wenn `true`, ist der Parameter auch ohne `default` optional (sonst HTTP 400 bei Fehlen).
`description`	Freitext-Beschreibung, die z. B. in der `serve`-UI angezeigt wird.

---

03 `select`: statisch vs. dataset-gestützt

03.01 Statische Liste (aus Kapitel 201)

```
queryParams:
  - name: REGION
    type: select
```

```
    options:
      items:
        - label: USA
          value: USA
        - label: Germany
          value: Germany
    default: USA
```

03.02 Aus einem DataSet

Statt jeden Wert von Hand zu pflegen, kannst du die Optionen direkt aus einem `DataSet` beziehen – praktisch, wenn sich die Werte (z. B. Regionen, Produktkategorien) ändern können:

```
apiVersion: bino.bi/v1alpha1
kind: DataSet
metadata:
  name: bn209_region_options
spec:
  query: >-
    SELECT DISTINCT rowGroup AS region_value, rowGroup AS region_label
    FROM bb_example_data
    ORDER BY rowGroup
  dependencies:
    - bb_example_data
```

```
queryParams:
  - name: REGION
    type: select
    options:
      dataset: bn209_region_options
      valueColumn: region_value
      labelColumn: region_label
    default: USA
```

Attribut	Beschreibung
`options.dataset`	Name des `DataSet`, aus dem die Auswahlliste geladen wird.
`options.valueColumn`	Spalte, deren Werte als Options-`value` verwendet werden.

Attribut	Beschreibung
`options.labelColumn`	Spalte für das angezeigte Label. Fehlt sie, wird `valueColumn` auch als Label genutzt.

---

04 `number` / `number_range`: Grenzen und Schrittweite

```
queryParams:
  - name: MIN_REVENUE
    type: number_range
    options:
      min: 0
      max: 1000
      step: 10
    default: "0"
```

`options.min` / `max` / `step` gelten für `number` und `number_range` gleichermaßen und steuern u. a. die Eingabe-Steuerelemente in der `serve`-Oberfläche.

---

05 Beispiel: Zwei Routen, ein Parameter, zwei Optionsquellen

Das Projekt-Beispiel `bn201_live` zeigt beide Varianten nebeneinander – dieselbe `LayoutPage` (`bn201_drill_country`) und derselbe Parametername (`REGION`), einmal mit statischer, einmal mit dataset-gestützter Optionsliste:

```
apiVersion: bino.bi/v1alpha1
kind: LiveReportArtefact
metadata:
  name: bn201_live
spec:
  title: BN201 Dashboard
  routes:
    /:
      artefact: handbuch_examples_bn201

    /drill: # statische Optionsliste
      layoutPages: bn201_drill_country
```

```
    queryParams:
      - name: REGION
        type: select
        options:
          items:
            - label: USA
              value: USA
            - label: Germany
              value: Germany
            # ...
        default: USA
      - name: YEAR
        type: date
        default: 2024-12-31T00:00:00Z

  /by-region: # dataset-gestützte Optionsliste
    layoutPages: bn201_drill_country
    queryParams:
      - name: REGION
        type: select
        options:
          dataset: bn209_region_options
          valueColumn: region_value
          labelColumn: region_label
        default: USA
      - name: YEAR
        type: date
        default: 2024-12-31T00:00:00Z
```

Gestartet mit:

```
bino serve --live bn201_live --port 45699
```

liefert `GET /by-region?REGION=Germany&YEAR=2024-12-31T00:00:00Z` dieselbe gefilterte Ansicht wie `/drill?REGION=Germany&...` – nur dass die Liste der wählbaren Regionen jetzt automatisch aus `bb_example_data` stammt, statt von Hand im Manifest gepflegt zu werden.

---

06 Zusammenfassung

1. **`routes`** ist eine Map von URL-Pfad zu Routen-Konfiguration; die Route `/` ist Pflicht.
2. Jede Route zeigt entweder ein **`artefact`** (ganzes `ReportArtefact`) oder **`layoutPages`** (eine oder mehrere Seiten) – nie beides.
3. **`queryParams`**-Typen: `string`, `number`, `number_range`, `select`, `date`, `date_time`.
4. **`select`**-Optionen kommen entweder aus einer statischen `items`-Liste oder – über `options.dataset` / `valueColumn` / `labelColumn` – direkt aus einem `DataSet`.
5. `bino serve --live <name>` rendert Routen bei Bedarf; Query-Parameter werden wie `${VAR}` in die referenzierten Manifeste eingesetzt.

> **Jetzt bist du dran!** Ergänze `bn201_live` um eine weitere Route mit einem `number_range`-Parameter (z. B. ein Mindestumsatz-Filter) und binde ihn über `filter` in `bn201_drill_country` ein.

210 - Hooks

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).

Kapitel 102 hat `[preview.hooks]` bereits kurz erwähnt (`pre-preview`, auskommentiert in `bino.toml`). Dieses Kapitel erklärt das komplette Hook-System: neun Checkpoints über vier `bino.toml` -Abschnitte, mit denen du eigene Shell-Skripte in den Build-, Preview- oder Serve-Ablauf einklinkst.

Du lernst:

- welche neun Hook-Checkpoints es gibt und in welchem `bino.toml` -Abschnitt sie stehen,
- welche Umgebungsvariablen ein Hook-Skript erhält,
- wie ein Hook sich selbst per Exit-Code `78` überspringen kann,
- und wie ein `[build.hooks]` -Eintrag in der Praxis aussieht.

---

01 Die neun Checkpoints

Checkpoint	Abschnitt in `bino.toml`	Zeitpunkt
`pre-build`	`[build.hooks]`	Nach dem Laden der Manifeste, bevor der Build beginnt.
`pre-datasource`	`[build.hooks]`	Bevor die Abfragen einer DataSource ausgeführt werden.
`pre-render`	`[build.hooks]`	Nach dem Erzeugen des HTML, bevor die PDF-Generierung startet (pro Artefakt).
`post-render`	`[build.hooks]`	Nachdem die PDF erzeugt (und ggf. signiert) wurde (pro Artefakt).
`post-build`	`[build.hooks]`	Nachdem alle Artefakte gebaut wurden.
`pre-preview`	`[preview.hooks]`	Einmalig, nachdem der Vorschau-Server gestartet ist, vor dem ersten Laden.

Checkpoint	Abschnitt in `bino.toml`	Zeitpunkt
`pre-refresh`	`[preview.hooks]`	Vor jedem Hot-Reload-Zyklus.
`pre-serve`	`[serve.hooks]`	Einmalig, nachdem der Serve-Server gestartet ist, vor der ersten Anfrage.
`pre-request`	`[serve.hooks]`	Vor jeder On-Demand-Render-Anfrage.

Jeder Hook ist ein Array von Shell-Befehlen (als Strings), die der Reihe nach ausgeführt werden:

```
[build.hooks]
pre-build = ["scripts/210_log_hook.sh"]
```

---

02 Umgebungsvariablen

Variable	Wann gesetzt
`BINO_MODE`	Immer (`build` , `preview` oder `serve`).
`BINO_WORKDIR`	Immer (Projekt-Wurzelverzeichnis).
`BINO_REPORT_ID`	Immer (aus `bino.toml`).
`BINO_HOOK`	Immer (Name des aktuellen Checkpoints, z. B. `pre-build`).
`BINO_VERBOSE`	Immer (`0` / `1`).
`BINO_ARTEFACT_NAME`	Nur bei artefakt-bezogenen Hooks (`pre-datasource` , `pre-render` , `post-render`).
`BINO_ARTEFACT_KIND`	Wie oben (`report` , `screenshot` , `document`).
`BINO_PDF_PATH`	Nur bei `post-render` .
`BINO_OUTPUT_DIR` , `BINO_INCLUDE` , `BINO_EXCLUDE`	Nur bei Build-Hooks.
`BINO_LISTEN_ADDR`	Nur bei Preview-/Serve-Hooks.
`BINO_REFRESH_REASON`	Nur bei `pre-refresh` .
`BINO_LIVE_ARTEFACT`	Nur bei Serve-Hooks.

`pre-build` und `post-build` laufen genau einmal pro `bino build`-Aufruf und erhalten deshalb **kein** `BINO_ARTEFACT_NAME` / `BINO_ARTEFACT_KIND` – diese beiden Variablen sind leer, wenn du sie dort ausliest.

---

03 Selbst überspringen: Exit-Code 78

Gibt ein Hook-Skript den Exit-Code `78` zurück, wertet BinoBI das **nicht** als Fehler, sondern protokolliert den Skip auf Info-Ebene und fährt fort. Praktisch für Hooks, die sich kontextabhängig selbst deaktivieren sollen:

```
#!/usr/bin/env bash
if [ "$BINO_MODE" != "build" ]; then
  exit 78 # nur im Build aktiv, in preview/serve überspringen
fi
# ... eigentliche Hook-Logik ...
```

---

04 Beispiel: Jeden Build-Checkpoint protokollieren

Das Projekt verdrahtet zu Demonstrationszwecken ein einfaches Logging-Skript an alle vier Build-Checkpoints:

```
# scripts/210_log_hook.sh
#!/usr/bin/env bash
echo "[$BINO_HOOK] mode=$BINO_MODE artefact=${BINO_ARTEFACT_NAME:-} kind=${BINO_
```

```
# bino.toml
[build.hooks]
pre-build = ["scripts/210_log_hook.sh"]
pre-render = ["scripts/210_log_hook.sh"]
post-render = ["scripts/210_log_hook.sh"]
post-build = ["scripts/210_log_hook.sh"]
```

Nach `bino build --artefact bn107_grid_pdf` enthält `dist/210_hook_log.txt` genau die erwarteten vier Zeilen – gut sichtbar, dass `pre-build` / `post-build` kein Artefakt kennen, `pre-render` / `post-render` aber schon:

```
[pre-build] mode=build artefact= kind=
[pre-render] mode=build artefact=bn107_grid_pdf kind=report
[post-render] mode=build artefact=bn107_grid_pdf kind=report
[post-build] mode=build artefact= kind=
```

---

05 Typische Anwendungsfälle

- **`pre-preview`**: Projekt-Abhängigkeiten aktuell halten, z. B. `bino update` (siehe der auskommentierte Beispiel-Eintrag in `bino.toml`, Kapitel 102).
- **`pre-build`**: Voraussetzungen prüfen (z. B. ob nötige Umgebungsvariablen für `ConnectionSecret`s gesetzt sind) und mit einer Fehlermeldung abbrechen, falls nicht.
- **`post-render`**: Eine erzeugte PDF automatisch in einen Ablageort hochladen oder eine Prüfsumme protokollieren (`BINO_PDF_PATH` zeigt auf die Datei).
- **`post-build`**: Team-Benachrichtigung (z. B. Slack-Webhook), dass ein neuer Bericht gebaut wurde.
- **`pre-request`**: Bei einem `LiveReportArtefact` z. B. Zugriff protokollieren oder Rate-Limiting umsetzen.

---

06 Zusammenfassung

1. Neun Checkpoints verteilen sich auf vier `bino.toml`-Abschnitte: `[build.hooks]`, `[preview.hooks]`, `[serve.hooks]` und projektweite `[hooks]`.
2. Jeder Hook ist ein Array von Shell-Befehlen, ausgeführt der Reihe nach.
3. Umgebungsvariablen wie `BINO_MODE`, `BINO_HOOK`, `BINO_ARTEFACT_NAME` / `BINO_ARTEFACT_KIND` geben dem Skript Kontext.
4. **Exit-Code `78`** lässt einen Hook sich selbst überspringen, ohne den Build als fehlgeschlagen zu markieren.
5. Artefakt-bezogene Hooks (`pre-datasource`, `pre-render`, `post-render`) laufen pro Artefakt; `pre-build` / `post-build` genau einmal pro Aufruf.

> **Jetzt bist du dran!** Ergänze `scripts/210_log_hook.sh` so, dass es bei `exit 78` abbricht, wenn `BINO_ARTEFACT_KIND` nicht `report` ist – und beobachte, dass der Hook beim `document`-Artefakt `handbuch` übersprungen wird.

211 – Streu- und Blasendiagramme: ChartScatter und ChartBubble

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/chart-scatter/ (https://cli.bino.bi/reference/chart-scatter/) und https://cli.bino.bi/reference/chart-bubble/ (https://cli.bino.bi/reference/chart-bubble/).

Bisher hattest du zwei Diagrammarten: `ChartStructure` vergleicht Kategorien (Kapitel 203), `ChartTime` zeigt Verläufe (Kapitel 204). Beide haben **eine** Wertachse.

`ChartScatter` und `ChartBubble` haben **zwei** – und beantworten damit eine andere Art von Frage: nicht „wie groß ist X?", sondern „hängen X und Y zusammen?". Marge gegen Umsatz, Wachstum gegen Planabweichung, Marktanteil gegen Marktwachstum. `ChartBubble` ergänzt eine dritte Kennzahl als Blasenfläche und wird damit zur klassischen Portfolio-Matrix.

Du lernst:

- wann ein Streudiagramm die richtige Wahl ist – und wann nicht,
- wie du `ChartScatter` mit `x` und `y` aufbaust,
- wie Achsen über `min`, `max`, `refLine` und `highlight` konfiguriert werden,
- wie Iso-Linien einen dritten, abgeleiteten Wert sichtbar machen,
- wie `ChartBubble` mit `size` eine dritte Kennzahl als Fläche darstellt,
- wie `share` Tortenanteile und `compareWith` ein Vergleichsszenario ergänzt,
- und welche Datenstruktur beide Komponenten erwarten.

> **Voraussetzung:** `ChartScatter`, `ChartBubble` und `ChartBullet` (Kapitel 212) benötigen eine Template-Engine ab **`v1.0.0-next.19`**. Mit älteren Engines – etwa der `v1.0.0-alpha.15` – validieren die Manifeste zwar und der Build läuft durch, das Diagramm bleibt aber **leer**. Prüfe im Zweifel `engine-version` in deiner `bino.toml` und aktualisiere mit `bino setup --template-engine --engine-version v1.0.0-next.22`.

---

01 Die Datenstruktur: eine Zeile, ein Punkt

Beide Komponenten folgen einer einfachen Regel:

> **Eine Zeile des DataSets ist genau ein Punkt bzw. eine Blase.**

Anders als bei `ChartStructure` wird **nicht** aggregiert. Was du zeichnen willst, musst du im DataSet fertig aggregieren – per `GROUP BY`.

Gebraucht werden:

- eine **Hierarchie-Spalte**, die den Punkt benennt (`rowGroup`, `category` oder `subCategory`, jeweils mit `…Index`),
- die **Messwert-Spalten** für die Achsen (`ac1` – `ac4`, `pp1` – `pp4`, `fc1` – `fc4`, `pl1` – `pl4`),
- optional `operation` (`'+'` / `'–'`).

Welche Spalte den Punkt identifiziert, steuert `level`. Der Standard `auto` nimmt die **tiefste befüllte** Ebene in der Reihenfolge `rowGroup` → `category` → `subCategory`.

> **Die häufigste Stolperfalle:** Punkte werden über ihren Namen identifiziert. Kommt derselbe `category`-Wert zweimal vor – etwa „Cloud" einmal für EMEA und einmal für APAC –, behält BinoBI **die erste Zeile und verwirft die zweite**. Das Diagramm zeigt dann stillschweigend zu wenige Punkte; der Build meldet keinen Fehler, weil die Entdopplung erst beim Rendern passiert. Sorge deshalb für eindeutige Namen (`Cloud EMEA`, `Cloud APAC`) oder nimm mit `level` eine Ebene, die eindeutig ist.

---

02 Das erste Streudiagramm

`x` und `y` sind Pflicht, alles andere optional:

```
apiVersion: bino.bi/v1alpha1
kind: ChartScatter
metadata:
  name: produkt_portfolio
spec:
  dataset: produkte
  chartTitle: "Marge vs. Umsatz"
  x:
    measure: ac1
    label: "Marge"
    unit: "% vom Umsatz"
    min: 0
    max: 40
  y:
```

```
    measure: ac2
    label: "Umsatz"
    unit: "mEUR"
    min: 0
  labels:
    points: auto
    max: 12
```

Das Ergebnis: sechs Punkte, eingefärbt nach `rowGroup`, mit automatisch platzierten, überschneidungsfreien Beschriftungen.

► **Zum Ausprobieren:** Streudiagramm in der Sandbox öffnen (https://sandbox.bino.bi/s/3ja3l5ucy1sak46) – forke es und setze z. B. `max: 40` auf `max: 20`, um zu sehen, wie Punkte außerhalb des Achsenbereichs behandelt werden.

> Quelle im Repo: 211_01_scatter_basis.yaml

02.01 Kurzform für Achsen

Braucht eine Achse nur den Messwert, genügt der bloße Token:

```
x: ac1          # identisch zu:  x: { measure: ac1 }
y: ac2
```

---

03 Achsen konfigurieren

Beide Achsen kennen dieselben Felder:

Feld	Typ	Beschreibung
`measure`	string	Pflichtfeld. Szenario-Slot (`ac1` … `pl4`) oder Varianz-Token (siehe Kapitel 307).
`label`	string	Achsenbeschriftung.
`unit`	string	Einheit, wird an das Label angehängt.
`min` / `max`	number	Fester Achsenbereich. Ohne Angabe skaliert BinoBI automatisch.
`refLine`	number	Referenzlinie **senkrecht zu dieser Achse** – bei `x` also eine vertikale Linie.

Feld	Typ	Beschreibung
`highlight`	object	Schattiertes Band entlang dieser Achse, z. B. `{ from: 30 }`.

Zwei Hinweise aus der Praxis:

- **`min` / `max` fixieren, sobald zwei Diagramme verglichen werden sollen.** Automatische Skalierung optimiert jedes Diagramm für sich – und macht sie damit untereinander unvergleichbar.
- **`refLine` ist das Werkzeug für Portfolios.** Eine Linie bei `refLine: 1` auf der x-Achse und bei `refLine: 5` auf der y-Achse teilt die Fläche in vier Quadranten – ohne dass du sie zeichnen musst.

---

04 Iso-Linien: die dritte Kennzahl ohne dritte Achse

Wenn das Produkt zweier Achsen eine sinnvolle Größe ergibt – Marge × Umsatz = Rohertrag –, macht `iso` diese Größe sichtbar. BinoBI zeichnet Hyperbeln `x · y = k`: Auf einer Linie liegen alle Kombinationen mit demselben Rohertrag.

```
spec:
  dataset: produkte
  x: { measure: ac1, label: "Marge", unit: "%",    min: 0, max: 40 }
  y: { measure: ac2, label: "Umsatz", unit: "mEUR", min: 0 }
  iso:
    values: [2000, 4000, 6000]
    label: "Rohertrag"
    unit: "mEUR"
    highlight: { from: 6000 }
```

`highlight.from` schattiert zusätzlich den Bereich, in dem `x · y >= 6000` gilt – die „guten" Produkte werden damit zu einer Fläche statt zu einer Rechenaufgabe.

Statt einer festen Liste akzeptiert `values` auch `auto`, dann wählt BinoBI passende Stufen.

> **Bedingung:** Iso-Linien brauchen einen **echt positiven** Wertebereich auf beiden Achsen. Mit Nullen oder negativen Werten ist `x · y = k` nicht sinnvoll darstellbar.

► **Zum Ausprobieren:** Iso-Linien in der Sandbox öffnen (https://sandbox.bino.bi/s/m6kdsb3i4s7ixji) – verschiebe `highlight: { from: 6000 }` auf `4000` und beobachte, wie sich das schattierte Feld ausdehnt.

> Quelle im Repo: 211_02_scatter_isolinien.yaml

---

05 ChartBubble: die dritte Kennzahl als Fläche

`ChartBubble` ist `ChartScatter` plus einer **Pflichtangabe** `size`. Der Wert bestimmt die **Fläche** der Blase – nicht den Radius. Das ist bewusst so: Die Fläche entspricht der Wahrnehmung, ein doppelter Wert wirkt doppelt so groß.

```
apiVersion: bino.bi/v1alpha1
kind: ChartBubble
metadata:
  name: bcg_portfolio
spec:
  dataset: geschaeftsfelder
  chartTitle: "Geschaeftsfeld-Portfolio"
  x:
    measure: ac1
    label: "Relativer Marktanteil"
    refLine: 1
  y:
    measure: ac2
    label: "Marktwachstum"
    unit: "%"
    refLine: 5
  size:
    measure: ac3
    label: "Umsatz"
    unit: "mEUR"
  seriesLevel: rowgroup
```

Mit den beiden `refLine`-Angaben entsteht die klassische Vier-Felder-Matrix; die Blasengröße zeigt, wie viel Umsatz in welchem Quadranten steckt.

Feld	Beschreibung
`size.measure`	Pflichtfeld. Werte müssen `>= 0` sein – negative Zeilen werden mit Warnung verworfen.
`size.label`	Beschriftung in der Größenlegende.

Feld	Beschreibung
`size.unit`	Einheit in der Größenlegende.
`size.group`	Name einer `ScalingGroup` für chartübergreifend vergleichbare Flächen (Kapitel 307).

► **Zum Ausprobieren:** Portfolio-Matrix in der Sandbox öffnen (https://sandbox.bino.bi/s/qob71e3sa1nery4) – ändere `refLine` auf der y-Achse und beobachte, wie sich die Quadranten verschieben.

> Quelle im Repo: 211_03_bubble_portfolio.yaml

---

06 Tortenanteile und Vergleichsszenario

Zwei Erweiterungen machen aus dem Portfolio ein Management-Chart:

`share` zeichnet einen Tortenanteil in die Blase – gedacht für Quoten wie „Anteil Wiederkehrende Umsätze". Die Werte sind **Bruchteile zwischen 0 und 1** (also `0.62`, nicht `62`); Werte außerhalb werden mit Warnung begrenzt. Der Anteil beginnt bei 12 Uhr.

`compareWith` legt ein zweites Blasen-Set des Vergleichsszenarios hinter das primäre – der Soll-Ist-Vergleich als Bild:

```
spec:
  x: { measure: ac1, label: "Relativer Marktanteil", refLine: 1 }
  y: { measure: ac2, label: "Marktwachstum", unit: "%", refLine: 5 }
  size: { measure: ac3, label: "Umsatz", unit: "mEUR" }
  share: ac4
  compareWith: pl
```

`compareWith` liest **dieselben Slot-Nummern der Vergleichsfamilie**: `x: ac1` wird gegen `pl1` verglichen, `y: ac2` gegen `pl2`, `size: ac3` gegen `pl3`. Diese Spalten müssen also im DataSet vorhanden sein.

Erlaubte Werte sind `ac`, `pp`, `fc` und `pl` – die Familie muss sich von der primären unterscheiden, und mit Varianz-Token auf den Achsen lässt sich `compareWith` nicht kombinieren.

► **Zum Ausprobieren:** Share und compareWith in der Sandbox öffnen (https://sandbox.bino.bi/s/kv2nu4ury8u1lk3) – nimm `compareWith: pl` heraus und vergleiche, wie viel

Information die Overlay-Blasen tragen.

> Quelle im Repo: 211_04_bubble_share_compare.yaml

---

07 Farben, Beschriftungen und Legende

Attribut	Standard (Scatter)	Standard (Bubble)	Beschreibung
`level`	`auto`	`auto`	Ebene, die den Punkt identifiziert: `auto`, `category`, `subcategory`.
`seriesLevel`	`auto`	`auto`	Ebene für die Farbserie: `auto`, `rowgroup`, `category`, `none`.
`labels`	`points: auto`, `values: false`, `max: 12`	`points: all`, `values: true`, `max: 12`	Beschriftung: `auto` platziert bis zu `max` überschneidungsfreie Labels.
`legend`	`show: true`, `position: right`	wie Scatter	Serienlegende; `position` ist `right` oder `bottom`.
`aspect`	`16:9`	`16:9`	Seitenverhältnis der Zeichenfläche – XY-Charts haben keine datengetriebene Höhe.
`limit`	`0`	`0`	Maximale Punkte je Panel. `0` = unbegrenzt.
`filter`	`""`	`""`	AlaSQL-`WHERE`-Fragment.

`seriesLevel` muss **echt oberhalb** von `level` liegen: Wenn `category` den Punkt benennt, kann die Farbe nach `rowGroup` gruppieren – umgekehrt nicht. `seriesLevel: none` zeichnet alles einfarbig.

Der Unterschied bei `labels` ist kein Zufall: In einem Streudiagramm können viele Punkte dicht beieinanderliegen, deshalb ist `auto` mit einer Obergrenze sinnvoll. Blasen sind größer, weiter auseinander und tragen die Beschriftung im Inneren – dort ist `all` die bessere Voreinstellung.

---

08 Welche Komponente wann?

Frage	Komponente
Wie groß ist der Umsatz je Region?	`ChartStructure`
Wie hat sich der Umsatz entwickelt?	`ChartTime`
Hängen Marge und Umsatz zusammen?	`ChartScatter`
Welche Geschäftsfelder tragen welchen Umsatz?	`ChartBubble`
Wie steht jede Kennzahl zu ihrem Ziel?	`ChartBullet` (Kapitel 212)

Ein Anhaltspunkt: Sobald die Frage zwei Kennzahlen **in Beziehung** setzt, ist ein XY-Chart richtig. Geht es um eine Kennzahl über eine Dimension, bleib bei `ChartStructure`.

---

09 Tipps

- **Im DataSet aggregieren, nicht im Chart.** Eine Zeile ist ein Punkt – ein vergessenes `GROUP BY` erzeugt stillschweigend zu wenige Punkte.
- **Eindeutige Punktnamen** sind Pflicht, sobald mehrere Gruppen dieselben Kategorienamen tragen (siehe Abschnitt 01).
- **`min` / `max` setzen**, sobald zwei Diagramme nebeneinander stehen – sonst täuscht die Auto-Skalierung Unterschiede vor.
- **`aspect` an das Layout anpassen:** In einer schmalen Karte ist `4:3` oft lesbarer als das voreingestellte `16:9`.
- **Bei vielen Punkten** hilft `labels.points: none` plus `legend` mehr als 40 überlappende Beschriftungen.

> Vollständige Referenz: cli.bino.bi/reference/chart-scatter/ (https://cli.bino.bi/reference/chart-scatter/) und cli.bino.bi/reference/chart-bubble/ (https://cli.bino.bi/reference/chart-bubble/)

212 – Ziel-Ist-Vergleiche: die ChartBullet-Komponente

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/chart-bullet/ (https://cli.bino.bi/reference/chart-bullet/).

Ein Management-Cockpit beantwortet immer dieselbe Frage: **Wo stehen wir gegenüber dem Ziel?** Umsatz gegen Plan, EBIT gegen Plan, Kosten gegen Budget – fünf Kennzahlen, fünf Zielwerte, ein Blick.

Genau dafür ist `ChartBullet` gebaut: pro Kennzahl eine Zeile mit einem Ist-Balken und einer Ziel-Markierung. BinoBI setzt dabei die **IBCS-Variante** des Bullet-Graphs um, die gegenüber dem Original von Stephen Few zwei Verbesserungen mitbringt – normalisierte Ziele und integrierte Abweichungsbalken.

Du lernst:

- wie ein Bullet-Graph aufgebaut ist und wofür er gedacht ist,
- warum die Normalisierung auf das Ziel Kennzahlen vergleichbar macht,
- wie `actual` und `target` automatisch erkannt werden,
- wie `operation` bestimmt, ob eine Abweichung gut oder schlecht ist,
- wie du Zeilen sortierst und begrenzt,
- und wie du bei Bedarf auf den klassischen Bullet-Graph umschaltest.

> **Voraussetzung:** `ChartBullet` benötigt eine Template-Engine ab **`v1.0.0-next.19`**. Mit älteren Engines validiert das Manifest, das Diagramm bleibt aber leer (siehe Kapitel 211).

---

01 Die Datenstruktur

Eine Zeile des Diagramms ist eine **Kennzahl**. Alle DataSet-Zeilen mit demselben Wert auf der gewählten Ebene werden dafür schlicht per `SUM` zusammengefasst.

Gebraucht werden:

- eine **Hierarchie-Spalte** (`rowGroup` , `category` oder `subCategory`) als Name der Kennzahl,

- eine **Ist-Spalte** – üblicherweise `ac1`,
- eine **Ziel-Spalte** – üblicherweise `pl1`,
- die Spalte `operation` mit `'+'` oder `'−'`.

`operation` ist hier kein Detail, sondern trägt die Bedeutung:

> `'−'` **kennzeichnet Kosten-Kennzahlen und dreht die Bewertung um.** Ein Ist **unter** dem Ziel ist bei Kosten *günstig* (grün), bei Erlösen *ungünstig* (rot). Ohne korrektes `operation` färbt das Diagramm Kostenüberschreitungen fälschlich grün.

---

02 Der erste Bullet-Graph

Im Minimalfall genügt das DataSet – alles Übrige wird erkannt:

```
apiVersion: bino.bi/v1alpha1
kind: ChartBullet
metadata:
  name: kpi_uebersicht
spec:
  dataset: kpis
  chartTitle: "KPI-Uebersicht vs. Plan"
```

BinoBI ermittelt dabei:

- `actual`: bevorzugt `ac1`, sonst das erste verfügbare Szenario,
- `target`: bevorzugt `pl1`, dann `pp1`, dann `fc1`,
- `level`: die tiefste befüllte Hierarchie-Ebene,
- **Sortierung**: die Reihenfolge aus dem DataSet.

Findet sich gar kein Ziel-Szenario, zeichnet BinoBI nur die Ist-Balken und warnt.

▶ **Zum Ausprobieren:** Bullet-Graph in der Sandbox öffnen (https://sandbox.bino.bi/s/yu3smvm1f7hyyx0) – im Beispiel sind „Personalkosten" und „Sachkosten" mit `operation: "−"` als Kosten markiert. Forke es, setze `operation` bei „Personalkosten" auf `"+"` und beobachte, wie der Abweichungsbalken von Rot auf Grün kippt, obwohl sich keine Zahl geändert hat.

> Quelle im Repo: 212_01_bullet_minimal.yaml

---

03 Warum Normalisierung der entscheidende Punkt ist

Im Standardmodus (`normalize: target`) wird **jede Zeile auf ihr eigenes Ziel bezogen**. Das Ziel liegt damit für alle Kennzahlen an derselben Stelle – bei 100 % – und aus vielen Einzelskalen wird **eine durchgehende Ziel-Linie**.

Der Nutzen wird an einem Zahlenbeispiel deutlich:

Kennzahl	Ist	Ziel	Zielerreichung
Umsatz	1.240	1.200	103 %
Rohertrag	486	520	93 %
EBIT	172	160	108 %
Personalkosten	214	200	107 % (Kosten)

Absolut sind diese Werte nicht vergleichbar – 1.240 gegen 172 sagt nichts aus. Normalisiert liegen alle vier auf derselben Skala, und die Frage „welche Kennzahl steht am schlechtesten?" ist auf einen Blick beantwortet.

Zeilen ohne oder mit nicht-positivem Ziel lassen sich nicht normalisieren; sie erscheinen als Label mit Wert und erzeugen eine Warnung.

---

04 Ist und Ziel explizit setzen

Sobald das DataSet mehrere Szenarien führt, lohnt die explizite Angabe:

```
spec:
  dataset: umsatz_je_region
  chartTitle: "Umsatz vs. Plan je Region"
  actual: ac1
  target:
    measure: pl1
    label: "Plan"
  order: ac1
  orderDirection: desc
  labels:
    decimals: 0
```

`actual` und `target` akzeptieren den bloßen Slot (`actual: ac1`) oder die Objektform mit `measure` , `label` und `unit` .

> **Varianz-Token sind hier nicht erlaubt.** Die Abweichung *ist* der Vergleich von Ist und Ziel – ein Token wie `dac1_pl1` hätte keine Bedeutung. (Auf Achsen von `ChartScatter` ist das anders, siehe Kapitel 307.)

`unit` wird übrigens **einmal im Kopf** ausgegeben („in EUR k"), nicht an jedem Wert – das hält die Zeilen ruhig.

► **Zum Ausprobieren:** Sortierten Bullet-Graph in der Sandbox öffnen (https://sandbox.bino.bi/s/4hu3w56zwqsmluo) – setze `orderDirection` auf `asc` , um die schwächsten Regionen nach oben zu holen.

> Quelle im Repo: 212_02_bullet_ziel_sortierung.yaml

---

05 Attribute im Überblick

Attribut	Typ	Standard	Beschreibung
`dataset`	string / array	–	Pflichtfeld.
`actual`	string / object	`ac1`	Ist-Wert. Nur einfache Szenario-Slots.
`target`	string / object	`pl1` > `pp1` > `fc1`	Ziel-Wert. Nur einfache Szenario-Slots.
`normalize`	string	`target`	`target` (IBCS, normalisiert) oder `none` (klassisch, eigene Skala je Zeile).
`variances`	string	`auto`	`auto` zeigt Abweichungsbalken im normalisierten Modus, `none` unterdrückt sie.
`ranges`	array	–	Bis zu zwei aufsteigende Bruchteile des Ziels als graue Bänder. Nur im klassischen Modus.
`level`	string	`auto`	Ebene der Kennzahl: `auto` , `rowgroup` , `category` , `subcategory` .

Attribut	Typ	Standard	Beschreibung
`order`	string	`auto`	Sortierschlüssel: `auto`, eine Hierarchie-Spalte oder ein Szenario-Slot.
`orderDirection`	string	`asc`	`asc` oder `desc`.
`limit`	integer	`0`	Maximale Zeilenzahl. `0` = unbegrenzt.
`labels`	object	`show: auto`, `decimals: 1`	Wertbeschriftung; `decimals` zwischen 0 und 4.
`filter`	string	`""`	AlaSQL-`WHERE`-Fragment.
`chartTitle`	string	`""`	Titel über dem Diagramm.
`scale`	string / number	auto	Proportionale Skalierung; Bullet-Graphen passen sich an die Höhe des Containers an.
`selectedStyle`	string	–	Name eines `ComponentStyle` (Kapitel 207).
`ruleset`	string	–	Name eines `RuleSet` oder `inherited-closest` / `inherited-page` (Kapitel 213).

> **`limit` schneidet ab, ohne eine Restzeile zu bilden.** Das ist Absicht: Die Summe aus Umsatz, Marge und Mitarbeiterzahl wäre bedeutungslos. Anders als bei einer Tabelle gibt es hier also keine „Sonstige"-Zeile.

---

06 Der klassische Bullet-Graph

Wer die Darstellung nach Stephen Few braucht – etwa weil ein Berichtsstandard sie vorschreibt –, schaltet mit `normalize: none` um:

```
spec:
  dataset: umsatz_je_region
  chartTitle: "Umsatz vs. Plan (klassischer Bullet)"
  actual: ac1
  target: pl1
  normalize: none
  ranges: [0.6, 0.9]
```

```
  variances: none
  labels:
    decimals: 0
```

Das ändert drei Dinge: absolute Werte auf **eigener Skala je Zeile**, Skalenstriche mit Ziel-Tick pro Zeile, und qualitative graue Bänder über `ranges`.

`ranges` sind aufsteigende **Bruchteile des Ziels** – `[0.6, 0.9]` bedeutet „bis 60 % schwach, bis 90 % mittel, darüber gut". Mehr als zwei Bänder sind nicht möglich: Ein drittes Band hätte keine definierte Obergrenze mehr, und IBCS lehnt es aus gutem Grund ab.

> Im normalisierten Modus wird `ranges` ignoriert und BinoBI warnt – Bänder als Bruchteile des Ziels ergeben keinen Sinn, wenn das Ziel ohnehin überall bei 100 % liegt.

► **Zum Ausprobieren:** Klassischen Bullet-Graph in der Sandbox öffnen (https://sandbox.bino.bi/s/j7swjkecbxqjswh) – stelle `normalize` auf `target` zurück und vergleiche beide Darstellungen direkt.

> Quelle im Repo: 212_03_bullet_klassisch.yaml

---

07 IBCS-Modus oder klassisch?

Kriterium	`normalize: target` (Standard)	`normalize: none` (klassisch)
Skala	Bruchteile des Ziels	Absolutwerte je Zeile
Ziel-Linie	Eine durchgehende Linie	Ein Tick je Zeile
Kennzahlen untereinander vergleichbar	Ja	Nein
Abweichungsbalken	Ja (`variances: auto`)	Nein
Graue Qualitätsbänder	Nein	Ja (`ranges`)
Typischer Einsatz	Management-Cockpit	Vorgegebener Berichtsstandard

Für neue Berichte ist der Standardmodus fast immer die richtige Wahl – er beantwortet die eigentliche Frage („welche Kennzahl reißt ihr Ziel?") ohne Kopfrechnen.

---

08 Typische Fehler

Symptom	Ursache	Lösung
Diagramm bleibt leer	Template-Engine älter als `v1.0.0-next.19`	`bino setup --template-engine --engine-version v1.0.0-next.22`
Kostenüberschreitung erscheint grün	`operation` fehlt oder steht auf `'+'`	Kosten-Zeilen mit `operation: "-"` kennzeichnen
Eine Zeile hat keinen Balken	Ziel fehlt oder ist `<= 0`	Ziel-Spalte prüfen oder Zeile herausfiltern
`ranges` hat keine Wirkung	Diagramm läuft im normalisierten Modus	`normalize: none` setzen
Zu viele Zeilen	DataSet nicht gefiltert	`filter` oder `limit` setzen

---

09 Tipps

- **`operation` zuerst prüfen.** Es ist die häufigste Ursache für ein Diagramm, das plausibel aussieht und trotzdem falsch ist.
- **`order: ac1` mit `orderDirection: desc`** stellt die größten Kennzahlen nach oben – für ein Cockpit meist die richtige Leserichtung. Für ein Problem-Cockpit ist `asc` auf der Zielerreichung oft besser.
- **`labels.decimals: 0`** bei großen Zahlen: Nachkommastellen bei Millionenbeträgen kosten Platz und bringen nichts.
- **`ChartBullet` statt Tabelle**, wenn nur die Zielerreichung interessiert – eine Tabelle mit Ist, Ziel, Abweichung und Prozent braucht vier Spalten für dieselbe Aussage.

> Vollständige Referenz: cli.bino.bi/reference/chart-bullet/ (https://cli.bino.bi/reference/chart-bullet/)

213 – Szenario-Regeln steuern: das RuleSet-Manifest

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ruleset/ (https://cli.bino.bi/reference/ruleset/).

BinoBI weiß von sich aus, wie Szenarien darzustellen sind: `AC` erscheint als gefüllter dunkler Balken, `PL` umrandet, `PP` hellgrau, und die Spalten stehen in der IBCS-Reihenfolge Vorjahr → Plan → Forecast → Ist. Das ist kein Zufall, sondern der IBCS-Notationsstandard – und in den allermeisten Fällen genau richtig.

Manchmal passt es aber nicht: Euer Haus liest den Plan lieber ganz rechts, oder der Forecast soll direkt neben dem Ist stehen. Statt diese Regeln in jeder Komponente einzeln zu überschreiben, gibt es dafür ein eigenes Manifest: **`RuleSet`**.

Du lernst:

- welche drei Eigenschaften ein RuleSet je Szenario steuert,
- wie du die Spaltenreihenfolge über `sortIndex` änderst,
- wie `colorIndex` die IBCS-Farbcodierung steuert,
- wie `_default` global und benannte RuleSets gezielt wirken,
- wie Komponenten das RuleSet ihrer Seite erben,
- und welche eingebauten Werte du überschreibst.

---

01 Was ein RuleSet steuert

Ein RuleSet ordnet jedem **Szenario-Präfix** (`ac` , `fc` , `pl` , `bu` , `pp` , `py` sowie den Varianz-Präfixen) bis zu drei Eigenschaften zu:

Eigenschaft	Wirkung
`sortIndex`	Position des Szenarios in der Spalten- bzw. Serienreihenfolge.
`colorIndex`	IBCS-Farbcodierung: `10` gefüllt dunkel, `20` hellgrau, `30` schraffiert, `50` umrandet.
`name`	Anzeigename des Szenarios.

> **Wichtige Einschränkung zu `name`:** In der aktuell ausgelieferten Template-Engine (geprüft mit `v1.0.0-next.22`) wirkt sich `name` **nicht** auf die Spaltenüberschriften von Tabellen aus. Die dort sichtbaren Kürzel (`AC`, `PY`, `PL`, `FC`) stammen aus dem Internationalisierungs-Store und nicht aus dem RuleSet. Verlass dich für Umbenennungen also nicht auf `name` – `sortIndex` und `colorIndex` wirken dagegen zuverlässig. Die technischen Hintergründe stehen in Kapitel 308, Abschnitt 05.

Jedes Präfix wird automatisch auf **vier Varianten** ausgedehnt: `ac` erzeugt `ac1` bis `ac4`. Die Varianten erben die Basiswerte, wobei zum `sortIndex` jeweils `n * 10` addiert wird – `ac1` bis `ac4` bleiben also untereinander in Reihenfolge.

---

02 Die eingebauten Standardwerte

Bevor du etwas änderst, lohnt der Blick auf die Ausgangslage. Diese Werte bringt BinoBI als `_system`-Ebene mit:

Präfix	Name	`colorIndex`	`sortIndex`
`pp` / `py`	PP / PY	20	100
`pl` / `bu`	PL / BU	50	200
`fc`	FC	30	300
`ac`	AC	10	400
`dac` … `dpy`	dAC…dPY	wie Basis	Basis + 1000
`drac` … `drpy`	drAC…drPY	wie Basis	Basis + 2000

Daraus ergibt sich die IBCS-Lesereihenfolge **PP (100) < PL (200) < FC (300) < AC (400)**, gefolgt von absoluten und dann relativen Abweichungen.

---

03 Die Spaltenreihenfolge ändern

Angenommen, der Plan soll hinter dem Ist stehen. Dazu genügt ein einziger Wert – ein `sortIndex` oberhalb von 400:

```
apiVersion: bino.bi/v1alpha1
kind: RuleSet
metadata:
```

```
  name: corporateRules
spec:
  content:
    scenarios:
      pl:
        sortIndex: 900
```

Die Komponente meldet sich über `spec.ruleset` an:

```
apiVersion: bino.bi/v1alpha1
kind: Table
metadata:
  name: umsatz_tabelle
spec:
  dataset: umsatz
  type: sum
  sumTitle: "Umsatz gesamt"
  ruleset: corporateRules
```

Aus `PY | PL | AC` wird damit `PY | AC | PL`.

> **Überschreiben ist partiell.** Im Beispiel steht nur `sortIndex` – `colorIndex` und alles Übrige behalten ihre eingebauten Werte. Du beschreibst also immer nur die Abweichung, nie das gesamte Regelwerk.

► **Zum Ausprobieren:** Sortierung in der Sandbox öffnen (https://sandbox.bino.bi/s/i1pgbkgvk22d0fe) – forke das Beispiel und entferne `ruleset: corporateRules` aus der Tabelle. Die Spalten springen zurück auf die IBCS-Reihenfolge `PY | PL | AC`.

> Quelle im Repo: 213_01_ruleset_sortierung.yaml

> **Achtung:** Setzt eine Tabelle ihre Spalten explizit über `scenarios: ["pp1", "ac1", "pl1"]`, gewinnt diese Liste – der `sortIndex` steuert dann nichts mehr. RuleSets wirken auf die **automatisch** ermittelte Reihenfolge.

---

04 `_default` oder benannt?

Der `metadata.name` entscheidet über die Reichweite:

Name	Wirkung
`_default`	Gilt **automatisch für alle** Komponenten – ohne `ruleset`-Attribut.
beliebig	Gilt nur für Komponenten, die es über `spec.ruleset` referenzieren.

Für eine hausweite Konvention ist `_default` das richtige Werkzeug: einmal definiert, gilt sie überall, und niemand muss daran denken, sie zu referenzieren.

```
apiVersion: bino.bi/v1alpha1
kind: RuleSet
metadata:
  name: _default
spec:
  content:
    scenarios:
      fc:
        sortIndex: 350      # Forecast direkt vor das Ist
```

Benannte RuleSets sind die Ausnahme für einzelne Seiten – etwa einen Plan-Review, der die Plan-Spalte anders anordnet als der Rest des Berichts. Beide Ebenen greifen ineinander; wie genau, zeigt Kapitel 308.

---

05 Vererbung: einmal setzen, überall nutzen

`ruleset` gibt es nicht nur auf Komponenten, sondern auch auf `LayoutPage` und `LayoutCard`. Zusammen mit zwei Schlüsselwörtern erspart das viel Wiederholung:

Wert	Bedeutung
`inherited-page`	Übernimmt das RuleSet der umgebenden `LayoutPage` (Karten werden übersprungen).
`inherited-closest`	Übernimmt das RuleSet der **nächstgelegenen** `LayoutCard` oder `LayoutPage`.

```
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: seite
```

```
spec:
  ruleset: corporateRules
  titleScenarios: ["ac1", "pl1"]
  pageLayout: split-horizontal
  children:
    - kind: Table
      ref: umsatz_tabelle          # spec.ruleset: inherited-page
    - kind: ChartStructure
      ref: umsatz_chart            # spec.ruleset: inherited-page
```

Ein auf der Seite gesetztes RuleSet gilt außerdem für die **Titelzeile** der Seite – Szenario-Angaben in Kopfzeilen folgen damit automatisch denselben Regeln wie die Komponenten darunter.

► **Zum Ausprobieren:** Vererbung in der Sandbox öffnen (https://sandbox.bino.bi/s/k755jgrlsmqdg7k) – Tabelle und Diagramm holen sich beide `inherited-page`. Ändere den `sortIndex` im RuleSet einmal und beobachte, dass beide Komponenten folgen.

> Quelle im Repo: 213_02_ruleset_vererbung.yaml

---

06 `colorIndex`: die Farbcodierung

`colorIndex` bestimmt die IBCS-Behandlung einer Fläche:

Wert	Darstellung	Übliches Szenario
`10`	gefüllt dunkel	`AC` (Ist)
`20`	hellgrau	`PP` / `PY`
`30`	schraffiert	`FC`
`50`	umrandet (offen)	`PL` / `BU`

Wer den Plan schraffiert statt umrandet darstellen will, ändert genau einen Wert:

```
apiVersion: bino.bi/v1alpha1
kind: RuleSet
metadata:
  name: planSchraffiert
spec:
```

```
  content:
    scenarios:
      pl:
        colorIndex: 30
```

> **Gut zu wissen:** `colorIndex` wird in der aktuellen Engine von den XY-Komponenten ausgewertet – `ChartScatter`, `ChartBubble` und `ChartBullet` (Kapitel 211 und 212). Tabellen und `ChartStructure` leiten ihre Behandlung stattdessen aus der Szenario-Familie ab. Welche Komponente welche Eigenschaft liest, ist in Kapitel 308, Abschnitt 04 vollständig aufgeschlüsselt.

► **Zum Ausprobieren:** Farbcodierung in der Sandbox öffnen (https://sandbox.bino.bi/s/bd5sqiwpfwkbngw) – setze `colorIndex` auf `20` oder `50` und vergleiche die Ziel-Markierung.

> Quelle im Repo: 213_03_ruleset_colorindex.yaml

---

07 Die `fallback`-Regel

Neben `scenarios` kennt ein RuleSet den Abschnitt `fallback` – er greift für alle Szenario-Namen, die keinen eigenen Eintrag haben. Das ist relevant, sobald ein DataSet eigene Präfixe mitbringt:

```
spec:
  content:
    scenarios:
      pl:
        sortIndex: 900
    fallback:
      name: "Sonstige"
      colorIndex: 20
      sortIndex: 120
```

Eigenschaft	Standard	Beschreibung
`name`	`Series`	Anzeigename für unbekannte Szenarien.
`group`	`Series`	Gruppenbezeichnung.

Eigenschaft	Standard	Beschreibung
`colorIndex`	`10`	Farbcodierung.
`sortIndex`	`120`	Einsortierung – standardmäßig knapp nach `PP`.

---

08 Die JSON-Schreibweise

`content` akzeptiert neben einem YAML-Objekt auch einen JSON-String. Das ist praktisch, wenn eine Konfiguration aus einem Frontend oder einer anderen Quelle kommt und unverändert übernommen werden soll:

```
apiVersion: bino.bi/v1alpha1
kind: RuleSet
metadata:
  name: compactRules
spec:
  content: >-
    {"scenarios":{"pl":{"sortIndex":900}}}
```

Beide Formen sind gleichwertig. Für handgeschriebene Manifeste ist die YAML-Variante lesbarer.

---

09 Wo `ruleset` verfügbar ist

Manifest	`spec.ruleset`
`Table`	ja
`ChartStructure`	ja
`ChartTime`	ja
`ChartScatter`	ja
`ChartBubble`	ja
`ChartBullet`	ja
`LayoutPage`	ja

Manifest	`spec.ruleset`
`LayoutCard`	ja

---

10 Tipps

- **Erst `_default`, dann benannt.** Was für den ganzen Bericht gilt, gehört in `_default`; benannte RuleSets sind für begründete Ausnahmen.
- **Nur Abweichungen schreiben.** Ein RuleSet, das alle sechs Szenarien vollständig auflistet, ist meist ein Zeichen dafür, dass die eingebauten Werte gar nicht geprüft wurden.
- **`sortIndex` großzügig wählen.** Werte wie `900` statt `401` lassen Platz, später noch etwas dazwischen zu schieben.
- **Explizite `scenarios`-Listen und RuleSets** vertragen sich nicht gut – entscheide dich je Komponente für einen der beiden Wege.
- **IBCS ist eine Konvention, kein Zwang** – aber wenn du davon abweichst, tu es einmal zentral und nicht verstreut über zwanzig Komponenten.

> Vollständige Referenz: cli.bino.bi/reference/ruleset/ (https://cli.bino.bi/reference/ruleset/)

214 – Bausteine teilen: die bino Registry

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/guides/registry-and-dependencies/ (https://cli.bino.bi/guides/registry-and-dependencies/) und https://cli.bino.bi/cli/registry/ (https://cli.bino.bi/cli/registry/).

Sobald mehrere Berichtsprojekte im Haus existieren, wiederholt sich einiges: dasselbe Corporate-Design als `ComponentStyle`, dasselbe `RuleSet`, dieselbe Deckblatt-Karte. Solche Bausteine per Copy-and-paste zu verteilen funktioniert genau bis zur ersten Änderung.

Die **Registry** löst das wie ein Paketmanager: Bausteine werden als Pakete veröffentlicht, in `bino.toml` als Abhängigkeit deklariert, in `bino.lock` exakt festgenagelt und mit `bino registry install` reproduzierbar wiederhergestellt.

Du lernst:

- welche Dateien beim Arbeiten mit Paketen entstehen und welche ins Git gehören,
- wie du Abhängigkeiten hinzufügst, aktualisierst und entfernst,
- wie der Lockfile-Workflow reproduzierbare Builds sicherstellt,
- wie du installierte Komponenten per `ref` einbindest und parametrisierst,
- wie du eine private Registry konfigurierst und dich authentifizierst,
- und was beim Zusammenspiel mit lokalen Dokumenten zu beachten ist.

---

01 Die beteiligten Dateien

Datei / Verzeichnis	Rolle	Ins Git?
`bino.toml` `[dependencies]`	Wovon du abhängst (Name → Version oder Tag)	**ja**
`bino.lock`	Exakt aufgelöste Versionen und Prüfsummen	**ja – unbedingt**
`.bino/registry/`	Die materialisierten Paketdateien	**nein** – gitignoren

Das Muster kennst du aus anderen Ökosystemen: Die Deklaration sagt, *was* du willst; der Lockfile hält fest, *was daraus konkret wurde*; das Installationsverzeichnis ist jederzeit

wiederherstellbar.

`bino build`, `bino preview`, `bino serve` und der LSP beziehen `.bino/registry/` automatisch in die Manifest-Menge ein. Installierte Pakete verhalten sich damit wie lokale Dokumente – du musst nichts importieren.

---

02 Abhängigkeiten deklarieren

Pakete heißen immer `@scope/name`:

```
[dependencies]
"@acme/quarterly-report" = "1.2.3"    # exakte Version -- bleibt, bis du sie änd
"@acme/base-layout"      = "latest"   # Tag -- wandert bei 'bino registry update
```

Die Werte sind entweder eine **exakte Version** oder ein **Tag** (`latest`, `stable`, …). **Versionsbereiche gibt es bewusst nicht** – ein Bericht, der heute rendert, soll nächstes Quartal identisch rendern.

Von Hand bearbeitest du diese Tabelle selten. Dafür gibt es Kommandos:

```
bino registry add @acme/base-layout              # folgt "latest"
bino registry add @acme/quarterly-report@1.2.3   # pinnt die Version
bino registry remove @acme/base-layout           # entfernt samt ungenutzter Tra
```

`add` löst dabei die **transitive Hülle** auf – genau eine Version je Paket. Widersprüchliche Pins brechen mit einem Dependency-Conflict ab, statt still eine Variante zu bevorzugen. Anschließend lädt `add` die Dokumente, prüft ihre Digests, schreibt sie nach `.bino/registry/<scope>/<name>.yml` und trägt alles in `bino.lock` ein.

---

03 Der Lockfile-Workflow

Kommando	Wirkung
`bino registry install`	Stellt `.bino/registry/` **exakt wie gelockt** wieder her, ohne neu aufzulösen.
`bino registry update`	Löst Tag-Abhängigkeiten neu auf und schreibt den Lockfile fort. Exakte Pins bleiben unangetastet.

Kommando	Wirkung
`bino registry verify`	Prüft die installierten Dateien gegen die gelockten Digests.
`bino registry list`	Zeigt die installierten Pakete.
`bino registry info`	Zeigt die aufgelösten Metadaten eines Pakets.
`bino registry search`	Sucht Pakete in der Registry.

`install` ist das Kommando für **frische Clones und CI**. Es verweigert die Arbeit, wenn der Lockfile fehlt oder von `[dependencies]` abgewichen ist, und verweist dann auf `update` – so kann eine Pipeline nicht versehentlich etwas anderes bauen als geplant.

`verify` hasht die **kanonische Dokumentform**. Eine Datei neu zu formatieren bricht die Prüfung deshalb nicht; eine inhaltliche Änderung schon.

Ein typischer CI-Ablauf:

```
bino registry install   # gelockten Stand herstellen
bino registry verify    # bei manipulierten Paketen abbrechen
bino build
```

> **Commit den Lockfile.** Ohne `bino.lock` ist ein Checkout nicht reproduzierbar – dann entscheidet der Zeitpunkt des `install`, welche Version eines Tags ihr bekommt.

---

04 Installierte Komponenten verwenden

Pakete behalten ihren Namen mit Scope als `metadata.name`. Genau dieser Name wird zur Referenz:

```
apiVersion: bino.bi/v1alpha1
kind: LayoutPage
metadata:
  name: kommentarseite
spec:
  children:
```

```
    - kind: Text
      ref: "@acme/regionskommentar"
      params:
        REGION: "EU"
```

Das funktioniert für alle referenzierbaren Komponenten: `Text`, `Table`, `ChartStructure`, `ChartTime`, `Tree`, `Grid`, `LayoutCard` und `Image`.

`params` verhält sich exakt wie bei Seitenparametern (Kapitel 109): Das Paket deklariert seine Parameter in `metadata.params` und nutzt `${NAME}`-Platzhalter im Spec.

```
apiVersion: bino.bi/v1alpha1
kind: Text
metadata:
  name: "@acme/regionskommentar"
  params:
    - name: REGION
      type: string
      default: "EU"
spec:
  value: "Kommentar zur Region ${REGION}"
```

Vier Regeln lohnen sich zu merken:

- **Rangfolge:** explizite `params` schlagen deklarierte Defaults, die wiederum eine gleichnamige Umgebungsvariable schlagen.
- **`select`-Parameter** stellen wie gewohnt `${NAME_LABEL}` bereit.
- **Werte sind Strings** – auch Zahlen: `JAHR: "2026"`.
- **Inline-`spec`-Felder** am Kind wirken weiterhin als Override und werden **nach** der Parameterersetzung gemischt.

> **Wichtig für Paketautor:innen:** Nur `${NAME}`-Platzhalter, die als Parameter in `metadata.params` deklariert sind, bleiben für die Ersetzung erhalten. Jede andere `${VAR}` in einem Paket wird schon beim Laden aus der **Umgebung** aufgelöst – was in der Umgebung der Nutzenden fehlt, wird zum Problem. Deklariere darum konsequent alles, was konfigurierbar sein soll.

---

05 Namenskonflikte

Ein Paket, dessen Name mit einem lokalen Dokument **gleicher Art** kollidiert, lässt die Duplikatsprüfung des Builds fehlschlagen.

Die Lösung ist immer, das **lokale** Dokument umzubenennen: Der Paketname ist Teil des Vertrags mit der Registry und darf sich nicht verschieben. Da Paketnamen den Scope tragen (`@acme/base-layout`), tritt der Fall in der Praxis nur auf, wenn ein lokales Dokument denselben Scope-Namen führt.

---

06 Registry konfigurieren

Die Registry-URL wird in dieser Reihenfolge bestimmt (der erste Treffer gewinnt):

1. Das Flag `--registry` (nur bei den Auth-Kommandos).
2. `bino.toml` → `[registry].url` – die Projekteinstellung, die mitcommittet wird.
3. Die Umgebungsvariable `BINO_REGISTRY_URL` .
4. `~/.bino/config.toml` → `[registry].url` – die Einstellung pro Maschine.
5. Die öffentliche Registry `https://registry.bino.bi` .

```
# Im Projekt, in bino.toml
[registry]
url = "https://registry.bino.bi"    # optional -- das ist der Standard
token = "${ACME_REGISTRY_TOKEN}"     # optional -- literal oder ${ENV_VAR}
```

```
# Pro Maschine, in ~/.bino/config.toml
[registry]
url = "https://registry.corp.example.com"
```

Die Projektdatei gewinnt immer gegen Umgebungsvariable und globale Konfiguration – ein ausgechecktes Projekt zieht damit verlässlich seine eigene Registry. Die globale Konfiguration hält ausschließlich die URL; Tokens liegen woanders (siehe unten). Eine fehlerhafte `~/.bino/config.toml` ist ein Fehler und wird nicht stillschweigend ignoriert.

---

07 Authentifizierung

Öffentliche Pakete lassen sich anonym beziehen. Für private Registries erzeugst du in der Web-Oberfläche unter *Settings* → *Tokens* ein Personal Access Token (Präfix `bino_pat_`) und meldest dich einmal an:

```
bino registry login     # interaktiv: Token einfügen
bino registry logout    # entfernt und widerruft die gespeicherte Anmeldung
```

Das Token landet in `~/.bino/credentials.json` (Modus `0600`, nach Registry-URL geschlüsselt) und wird pro Aufruf gegen ein kurzlebiges Session-Token getauscht. `bino cache clean --global` lässt diese Datei unangetastet.

Für CI führt der Weg über die Umgebung:

```
export BINO_REGISTRY_TOKEN="bino_pat_..."
bino registry install
```

Die Reihenfolge bei der Token-Auswahl:

1. `bino.toml` → `[registry].token` (ein `${ENV_VAR}` wird expandiert; eine nicht gesetzte Variable ist ein Fehler).
2. `BINO_REGISTRY_TOKEN`.
3. Die per `bino registry login` gespeicherte Anmeldung für diese URL.
4. Anonym.

> **Für CI ein eigenes Token anlegen** – nicht das persönliche wiederverwenden. Es lässt sich in der Web-Oberfläche gezielt widerrufen, ohne die eigene Arbeitsumgebung zu treffen.

---

08 Veröffentlichen

Die CLI ist **rein konsumierend** – ein `bino publish` gibt es nicht. Pakete werden über den Registry-Dienst selbst veröffentlicht.

Für die Praxis heißt das: Der Weg vom internen Baustein zum Paket führt über die Web-Oberfläche eurer Registry, nicht über das Berichtsprojekt.

---

09 Was sich gut als Paket eignet

Gut geeignet	Schlecht geeignet
`ComponentStyle` mit dem Corporate Design	Ein komplettes `ReportArtefact`

Gut geeignet	Schlecht geeignet
`RuleSet` mit hausweiten Szenario-Regeln	DataSources mit projektspezifischen Pfaden
`LayoutCard` für ein einheitliches Deckblatt	DataSets mit fachspezifischer Geschäftslogik
`Text`-Bausteine (Impressum, Disclaimer) mit Parametern	Alles, was nur ein einziges Projekt je braucht

Der Unterschied liegt in der Parametrisierbarkeit: Ein Baustein wird dann wiederverwendbar, wenn seine Variablen sauber als Parameter deklariert sind (Kapitel 109, Abschnitt 06). Ein Paket, das man nach dem Installieren erst anpassen muss, ist keins.

---

10 Tipps

- **Für Anwendungsprojekte pinnen**, für Bibliotheksentwicklung Tags folgen. `latest` in einem Quartalsbericht ist ein Reproduzierbarkeitsrisiko.
- **`bino registry verify` in die CI** aufnehmen – die Prüfsummen sind sonst nur Dekoration.
- **`.bino/registry/` gitignoren.** Eingecheckte Paketdateien geraten unweigerlich aus dem Tritt mit dem Lockfile.
- **`bino registry update` bewusst ausführen**, in einem eigenen Commit. Dann ist im Diff sichtbar, welche Version sich bewegt hat.
- **Bei Namenskonflikten** das lokale Dokument umbenennen – nie versuchen, den Paketnamen zu umgehen.

> Vollständige Referenz: cli.bino.bi/guides/registry-and-dependencies/
> (https://cli.bino.bi/guides/registry-and-dependencies/)

301 - Cloud-Datenquellen und Secrets

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).
>
> **Hinweis zu diesem Kapitel:** Dieses Projekt verwendet ausschließlich lokale CSV-Dateien als Datenquelle – die folgenden Beispiele sind daher **illustrativ** und nicht gegen echte Cloud-Infrastruktur getestet. Die YAML-Struktur selbst ist jedoch schema-geprüft (`bino lint`) und entspricht exakt der aktuellen `ConnectionSecret` -Referenz.

Kapitel 103 hat `ConnectionSecret` am Beispiel von PostgreSQL knapp gestreift. Dieses Kapitel zeigt die vollständige Typenpalette: Objektspeicher (S3, GCS, R2, Azure), HTTP und Hugging Face – sowie den entscheidenden Unterschied, wie diese beiden Kategorien referenziert werden.

Du lernst:

- welche acht `type` -Werte `ConnectionSecret` unterstützt,
- den Unterschied zwischen **explizit referenzierten** (Datenbank-)Secrets und **automatisch nach `scope` zugeordneten** (Objektspeicher-)Secrets,
- die Felder jedes Auth-Blocks,
- und wie `provider: credential_chain` die automatische Zugangsdaten-Erkennung aktiviert.

---

01 Acht Secret-Typen

```
apiVersion: bino.bi/v1alpha1
kind: ConnectionSecret
metadata:
  name: my_secret
spec:
  type: s3 # s3 | gcs | http | r2 | azure | postgres | mysql | huggingface
```

`type`	Auth-Block	Typischer Zweck
`postgres`	`postgres`	Zugangsdaten für `postgres_query` -DataSources.

`type`	Auth-Block	Typischer Zweck
`mysql`	`mysql`	Zugangsdaten für `mysql_query`-DataSources.
`s3`	`s3`	AWS S3 oder S3-kompatible Speicher.
`gcs`	`gcs`	Google Cloud Storage.
`r2`	`r2`	Cloudflare R2.
`azure`	`azure`	Azure Blob Storage.
`http`	`http`	HTTP(S)-Downloads, inkl. Proxy-Konfiguration.
`huggingface`	`huggingface`	Hugging-Face-Datasets/-Hub.

Der Auth-Block liegt jeweils unter einem Schlüssel **mit demselben Namen wie `type`** – nicht unter einem generischen `auth`-Feld.

---

02 Zwei grundlegend verschiedene Referenzierungs-Arten

02.01 Datenbank-Secrets: explizit über `connection.secret`

Bei `postgres_query` / `mysql_query`-DataSources verweist `spec.connection.secret` **explizit** auf den Namen des `ConnectionSecret`:

```
apiVersion: bino.bi/v1alpha1
kind: ConnectionSecret
metadata:
  name: warehouse_pg_secret
spec:
  type: postgres
  postgres:
    passwordFromEnv: PG_PASSWORD
---
apiVersion: bino.bi/v1alpha1
kind: DataSource
metadata:
  name: warehouse_sales
spec:
  type: postgres_query
  connection:
```

```
    host: db.example.internal
    port: 5432
    database: analytics
    schema: public
    user: reporting_ro
    secret: warehouse_pg_secret # <- explizite Referenz
  query: "SELECT * FROM sales_fact"
```

`connection` selbst enthält **nur** die nicht-geheimen Verbindungsdetails (`host` , `port` , `database` , `schema` , `user`); das Passwort kommt ausschließlich aus dem referenzierten Secret.

02.02 Objektspeicher-Secrets: automatisch über `scope`

Bei S3/GCS/R2/Azure/HTTP/Hugging-Face gibt es **kein** Referenzfeld auf der DataSource. Stattdessen ordnet DuckDB das passende Secret automatisch dem `path` einer Datei-DataSource zu, sofern dessen Präfix zu `spec.scope` passt:

```
apiVersion: bino.bi/v1alpha1
kind: ConnectionSecret
metadata:
  name: landing_zone_s3
spec:
  type: s3
  scope: "s3://acme-landing-zone" # gilt für alle Pfade mit diesem Präfix
  s3:
    keyIdFromEnv: AWS_ACCESS_KEY_ID
    secretFromEnv: AWS_SECRET_ACCESS_KEY
    region: eu-central-1
---
apiVersion: bino.bi/v1alpha1
kind: DataSource
metadata:
  name: fact_sales_parquet
spec:
  type: parquet
  path: "s3://acme-landing-zone/warehouse/fact_sales/*.parquet"
  # kein `secret:`-Feld nötig -- der Scope matcht automatisch
```

> **Warum der Unterschied?** Datenbankverbindungen sind 1:1 (eine `connection` – ein Secret), deshalb ist eine explizite Referenz eindeutig und sinnvoll. Objektspeicher-Pfade

> sind dagegen oft zahlreich und über viele DataSources verteilt – die Scope-basierte Zuordnung erspart es dir, dieselbe Referenz überall zu wiederholen.

---

03 Auth-Blöcke im Detail

03.01 `postgres` / `mysql`

```
postgres:
  passwordFromEnv: PG_PASSWORD # oder: password: "..." (nicht empfohlen, siehe u
```

Feld	Beschreibung
`password`	Passwort im Klartext im Manifest (nur für lokale Experimente).
`passwordFromEnv`	Name einer Umgebungsvariable, aus der das Passwort gelesen wird.

03.02 `s3`

```
s3:
  keyIdFromEnv: AWS_ACCESS_KEY_ID
  secretFromEnv: AWS_SECRET_ACCESS_KEY
  region: eu-central-1
  sessionTokenFromEnv: AWS_SESSION_TOKEN # optional, für temporäre Credentials
  endpoint: "https://s3.eu-central-1.example.com" # optional, für S3-kompatible
  urlStyle: path # path | vhost
```

03.03 `gcs` / `r2`

```
gcs:
  keyIdFromEnv: GCS_KEY_ID
  secretFromEnv: GCS_SECRET

r2:
  keyIdFromEnv: R2_KEY_ID
  secretFromEnv: R2_SECRET
  accountId: "1234567890abcdef"
  endpoint: "https://1234567890abcdef.r2.cloudflarestorage.com"
```

03.04 `azure`

```
azure:
  connectionStringFromEnv: AZURE_STORAGE_CONNECTION_STRING
  # Alternative: einzelne Felder statt Connection-String
  accountName: acmestorage
  accountKeyFromEnv: AZURE_STORAGE_ACCOUNT_KEY
```

03.05 `http` (inkl. Proxy)

```
http:
  bearerTokenFromEnv: API_BEARER_TOKEN
  # oder Basic-Auth:
  usernameFromEnv: API_USER
  passwordFromEnv: API_PASSWORD
  # Proxy-Konfiguration (unabhängig von der Authentifizierungsmethode oben):
  httpProxyFromEnv: HTTPS_PROXY
  httpProxyUsernameFromEnv: PROXY_USER
  httpProxyPasswordFromEnv: PROXY_PASSWORD
```

03.06 `huggingface`

```
huggingface:
  tokenFromEnv: HF_TOKEN
```

Jedes `...FromEnv`-Feld hat ein Gegenstück ohne `FromEnv`-Suffix (z. B. `secret` / `secretFromEnv`), das den Wert direkt im Manifest erwartet. **Bevorzuge in jedem Fall die `...FromEnv`-Variante** – damit landen Zugangsdaten nie im Git-Repository.

---

04 `provider: credential_chain`

Statt Zugangsdaten überhaupt im Manifest zu referenzieren, kannst du DuckDBs automatische Erkennung nutzen (z. B. IAM-Rollen, `~/.aws/credentials`, Umgebungsvariablen, die der jeweilige Cloud-Anbieter selbst vorgibt):

```
apiVersion: bino.bi/v1alpha1
kind: ConnectionSecret
metadata:
```

```
  name: landing_zone_s3_auto
spec:
  type: s3
  scope: "s3://acme-landing-zone"
  provider: credential_chain
```

Damit entfällt der komplette `s3:`-Auth-Block – nützlich, wenn `bino build` in einer Umgebung läuft, die bereits über eine IAM-Rolle oder AWS-SSO-Session verfügt (z. B. eine CI-Pipeline, siehe Kapitel 306).

---

05 Zusammenfassung

1. `ConnectionSecret.spec.type` kennt acht Werte: `postgres`, `mysql`, `s3`, `gcs`, `r2`, `azure`, `http`, `huggingface`.
2. Der Auth-Block liegt unter einem Schlüssel mit demselben Namen wie `type`.
3. **Datenbank-Secrets** werden explizit über `DataSource.spec.connection.secret` referenziert.
4. **Objektspeicher-/HTTP-Secrets** werden automatisch über `spec.scope` (Pfad-Präfix) zugeordnet – keine Referenz auf der DataSource nötig.
5. Nutze immer `...FromEnv`-Felder statt Klartext-Zugangsdaten im Manifest; `provider: credential_chain` erspart sogar das.

> **Jetzt bist du dran!** Skizziere ein `ConnectionSecret` vom Typ `azure` für einen fiktiven Speicher-Account, referenziere es über `scope` und binde eine `parquet`-DataSource mit einem `path: "az://..."` ein. Prüfe mit `bino lint`, ob die Struktur schema-valide ist – ganz ohne echten Zugriff auf Azure.

302 - DocumentArtefact: Markdown-Dokumente als PDF

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).

Alle bisherigen Kapitel drehten sich um die Pipeline DataSource → DataSet → LayoutPage → `ReportArtefact`. `DocumentArtefact` ist eine **zweite, unabhängige Rendering-Pipeline**: reines Markdown zu PDF, mit Inhaltsverzeichnis, LaTeX-Mathematik und Playwright-basierten Kopf-/Fußzeilen – ohne DuckDB, ohne Table/Chart-Komponenten. Das Handbuch, das du gerade liest, wird selbst über ein `DocumentArtefact` gebaut.

Du lernst:

- die Felder von `DocumentArtefact` (`sources`, Inhaltsverzeichnis, Mathematik, Kopf-/Fußzeilen),
- wie `reports/handbuch.yaml` dieses Handbuch selbst zusammenbaut,
- ein eigenes, kleineres Beispiel-Dokument,
- und eine beobachtete Einschränkung beim Formel-Rendering.

---

01 Grundstruktur

```
apiVersion: bino.bi/v1alpha1
kind: DocumentArtefact
metadata:
  name: my_document
spec:
  format: a4 # a4 | a5 | letter | legal
  orientation: portrait # portrait | landscape
  filename: my-document.pdf
  sources:
    - ./docs/*.md
```

Attribut	Pflicht	Beschreibung
`format`	ja	Seitenformat: `a4`, `a5`, `letter`, `legal`.
`sources`	ja	Geordnete Liste von Markdown-Dateipfaden oder Glob-Mustern.
`orientation`	nein	`portrait` (Standard) oder `landscape`.
`locale`	nein	Sprachcode für i18n/`bn-context` (Standard `de`).
`filename`	nein	Name der erzeugten PDF-Datei.
`title` / `author` / `subject` / `keywords`	nein	PDF-Metadaten.
`stylesheet`	nein	Pfad zu einem eigenen CSS-Stylesheet (relativ zur Manifest-Datei).
`signingProfile`	nein	Name eines `SigningProfile` zum digitalen Signieren (siehe Kapitel 304).

---

02 Inhaltsverzeichnis und Kapitelnummerierung

```
spec:
  tableOfContents: true # Standard: false
  tocNumbering: true # Standard: true, nur relevant wenn tableOfContents aktiv i
```

Das Inhaltsverzeichnis wird automatisch aus den Markdown-Überschriften (`#`, `##`, `###`, …) über **alle** `sources`-Dateien hinweg erzeugt, mit hierarchischer Nummerierung (`1`, `1.1`, `1.1.1`, …), sofern `tocNumbering` nicht explizit auf `false` gesetzt ist.

---

03 LaTeX-Mathematik mit KaTeX

```
spec:
  math: true # Standard: true
```

```
Inline: $E = mc^2$
```

```
Display:
$$
\int_0^\infty e^{-x} \, dx = 1
$$
```

> **Beobachtung:** In einem lokalen Test erschien neben der gerenderten Formel zusätzlich der unverarbeitete LaTeX-Ausdruck als Klartext (z. B. `A = πr² A = πr2`). Das kann versionsabhängig sein – prüfe Formeln vor dem produktiven Einsatz in `bino preview` mit deiner konkreten Engine-Version.

---

04 Kopf- und Fußzeilen

```
spec:
  displayHeaderFooter: true
  headerTemplate: '<div style="font-size:9px; width:100%; text-align:center;"><s
  footerTemplate: '<div style="font-size:9px; width:100%; text-align:center;"><s
  marginTop: "20mm" # Standard
  marginBottom: "15mm" # Standard
```

Verfügbare Playwright-Template-Klassen: `date`, `title`, `url`, `pageNumber`, `totalPages`. Ohne eigene Templates (aber mit `displayHeaderFooter: true`) zeigt BinoBI eine Standard-Kopf-/Fußzeile (Titel zentriert oben, Datum links/Seitenzahl rechts unten).

---

05 Wie dieses Handbuch selbst gebaut wird

```
# reports/handbuch.yaml
apiVersion: bino.bi/v1alpha1
kind: DocumentArtefact
metadata:
  name: handbuch
spec:
  format: a4
  orientation: portrait
  filename: binobi-handbuch.pdf
  title: "BinoBI Handbuch"
  author: "BinoBI Team"
```

```
  tableOfContents: true
  displayHeaderFooter: true
  marginTop: "20mm"
  marginBottom: "15mm"
  sources:
    - ../docs/manual/de/*.md
  pageBreakBetweenSources: true
```

Das Glob-Muster `../docs/manual/de/*.md` bindet **jede** Markdown-Datei in diesem Verzeichnis ein – inklusive dieses Kapitels. Neue Kapitel werden automatisch Teil des Handbuchs, sobald du die Datei anlegst; eine Aktualisierung von `sources` ist nicht nötig.

---

06 Ein eigenes, kleineres Beispiel

```
apiVersion: bino.bi/v1alpha1
kind: DocumentArtefact
metadata:
  name: bn302_document_example
spec:
  format: a4
  orientation: portrait
  filename: bn302_document_example.pdf
  title: "Kapitel 302 Beispiel-Dokument"
  tableOfContents: true
  tocNumbering: true
  math: true
  pageBreakBetweenSources: true
  displayHeaderFooter: true
  headerTemplate: '<div style="font-size:9px; width:100%; text-align:center;"><s
  footerTemplate: '<div style="font-size:9px; width:100%; text-align:center;"><s
  sources:
    - ../docs/examples/302/01_intro.md
    - ../docs/examples/302/02_details.md
```

`bino build --artefact bn302_document_example` erzeugt daraus ein zweiseitiges PDF: eine Inhaltsverzeichnis-Seite (eigene römische Seitenzahlen) plus zwei Inhaltsseiten (arabische Seitenzahlen `1 / 2`, `2 / 2`), mit Seitenumbruch zwischen den beiden Quelldateien.

---

07 Zusammenfassung

1. `DocumentArtefact` ist eine eigenständige Markdown-zu-PDF-Pipeline, unabhängig von DataSource/DataSet/LayoutPage.
2. `sources` ist eine geordnete Liste von Dateipfaden oder Glob-Mustern – **dieses Handbuch** wird per Glob aus `docs/manual/de/*.md` gebaut.
3. `tableOfContents` / `tocNumbering` erzeugen ein automatisches, hierarchisch nummeriertes Inhaltsverzeichnis.
4. `math: true` aktiviert KaTeX-Formeln (`$...$` / `$$...$$`) – teste das Ergebnis vor dem produktiven Einsatz.
5. `displayHeaderFooter` plus `headerTemplate` / `footerTemplate` (Playwright-Klassen `date` / `title` / `url` / `pageNumber` / `totalPages`) steuern Kopf- und Fußzeilen.

> **Jetzt bist du dran!** Erstelle ein eigenes `DocumentArtefact` , das zwei eigene Markdown-Dateien mit mindestens zwei Überschriftenebenen einbindet, und aktiviere `tableOfContents` . Prüfe, ob die Kapitelnummern im erzeugten PDF mit der Überschriften-Hierarchie übereinstimmen.

303 - ScreenshotArtefact

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).

`ScreenshotArtefact` exportiert einzelne Komponenten – keine ganzen Seiten – als PNG oder JPEG. Praktisch für Thumbnails, Social-Media-Previews oder visuelle Regressionstests einzelner Charts/Tabellen.

Du lernst:

- wie `refs` einzelne Komponenten adressiert,
- welche Bildformat- und Auflösungsoptionen es gibt,
- wie `filenamePattern` die Dateinamen steuert,
- und wie das Projekt-Beispiel `example_screenshots` aufgebaut ist.

---

01 Grundstruktur

```
apiVersion: bino.bi/v1alpha1
kind: ScreenshotArtefact
metadata:
  name: example_screenshots
spec:
  filenamePrefix: example
  refs:
    - kind: Table
      name: top_customers
```

Attribut	Pflicht	Beschreibung
`filenamePrefix`	ja	Präfix für alle erzeugten Dateinamen.
`refs`	ja	Liste von `{kind, name}`-Paaren, je eine pro gewünschtem Screenshot.

Attribut	Pflicht	Beschreibung
`layoutPages`	nein	Einschränkung auf bestimmte Seiten; ohne Angabe werden Seiten automatisch anhand der `refs` ermittelt.
`format`	nein	Logische Viewport-Größe, z. B. `xga`, `hd`, `full-hd`, `4k` (Standard `xga`).
`orientation`	nein	`portrait` / `landscape` (Standard `landscape`).
`language`	nein	`de` / `en` (Standard `de`).
`filenamePattern`	nein	`ref` (Standard, `prefix-refname.png`) oder `index` (`prefix-001.png`).
`imageFormat`	nein	`png` (Standard) oder `jpeg`.
`quality`	nein	JPEG-Qualität 1-100, nur bei `imageFormat: jpeg`.
`omitBackground`	nein	`true` für transparenten Hintergrund (nur PNG).
`scale`	nein	`css` oder `device` (Geräte-Pixel, z. B. für Retina-Auflösung).

`refs[].kind` erlaubt: `ChartStructure`, `ChartTime`, `Tree`, `Table`, `LayoutCard`, `Text`.

---

02 Beispiel aus dem Projekt

```
apiVersion: bino.bi/v1alpha1
kind: ScreenshotArtefact
metadata:
  name: example_screenshots
spec:
  filenamePrefix: example
  filenamePattern: ref
  imageFormat: png
  format: full-hd
  orientation: landscape
  language: de
  scale: device
  refs:
    - kind: Table
      name: top_customers
    - kind: Table
```

```
      name: umsatz_zahlen
    - kind: Table
      name: 202_example_grouped
```

`bino build --artefact example_screenshots` erzeugt daraus drei PNG-Dateien: `dist/example-top_customers.png`, `dist/example-umsatz_zahlen.png` und `dist/example-202_example_grouped.png`. Bemerkenswert: Die referenzierten Komponenten müssen dafür **nicht** Teil einer regulären `LayoutPage` sein, die auch als `ReportArtefact` gebaut wird – `umsatz_zahlen` und `202_example_grouped` (aus Kapitel 202) existieren ausschließlich als eigenständige Komponenten-Dateien und werden trotzdem korrekt eingefangen.

---

03 `filenamePattern`: `ref` vs. `index`

```
filenamePattern: ref # example-top_customers.png, example-umsatz_zahlen.png, ...
filenamePattern: index # example-001.png, example-002.png, ...
```

`ref` ist meist die bessere Wahl für Nachvollziehbarkeit; `index` eignet sich, wenn nachgelagerte Werkzeuge eine feste, nummerierte Reihenfolge erwarten.

---

04 Praktische Hinweise

- Nicht jede Kombination aus Komponenten-Art und Einbettungskontext liefert zuverlässig einen Screenshot – in einem lokalen Test schlug das Einfangen eines `LayoutCard` sowie eines `ChartTime`, die beide nur in Baustein-Dateien ohne zugehörige gerenderte Seite existierten, mit einem Timeout fehl. Teste neue `refs` einzeln, bevor du sie in eine größere Liste aufnimmst.
- `omitBackground: true` liefert transparente PNGs – praktisch, um einen Chart direkt auf einer eigenen Präsentationsvorlage zu platzieren.
- `scale: device` erzeugt schärfere Bilder auf hochauflösenden Displays, allerdings auch größere Dateien.

---

05 Zusammenfassung

1. **`ScreenshotArtefact`** exportiert einzelne Komponenten (nicht ganze Seiten) als PNG/JPEG.

2. `refs` ist eine Liste von `{kind, name}` – referenzierte Komponenten müssen nicht Teil eines regulären `ReportArtefact` sein.
3. `format` / `orientation` steuern die Render-Auflösung, `imageFormat` / `quality` / `omitBackground` / `scale` die Bildausgabe.
4. `filenamePattern: ref` (Standard) oder `index` steuern die Dateinamen.
5. Teste neue `refs` einzeln – nicht jede Kombination aus Komponenten-Art und Kontext liefert zuverlässig ein Bild.

> **Jetzt bist du dran!** Füge `example_screenshots` einen weiteren `Table`-Ref auf eine eigene, kleine Beispieltabelle hinzu, setze `imageFormat: jpeg` mit `quality: 80` und vergleiche die Dateigröße mit der PNG-Variante.

304 - Digitale Signatur von PDFs

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).
>
> **Wichtig:** Das Zertifikat in diesem Kapitel ist ein **selbstsigniertes Test-Zertifikat**, lokal mit `openssl` erzeugt, ausschließlich zu Demonstrationszwecken. Verwende es niemals für echte, rechtsverbindliche Signaturen – dafür brauchst du ein Zertifikat einer vertrauenswürdigen Zertifizierungsstelle (CA).

`SigningProfile` signiert die von `ReportArtefact` oder `DocumentArtefact` erzeugten PDFs digital (PAdES-artig): Signaturfeld, Signer-Metadaten, optionaler Zeitstempel-Dienst (TSA) und Berechtigungen für nachträgliche Änderungen.

Du lernst:

- wie ein `SigningProfile` aufgebaut ist,
- wie Zertifikat und privater Schlüssel eingebunden werden (`inline` vs. `path`),
- was `certType` und `docMDPPerm` bedeuten,
- und wie du eine echte, kryptographisch gültige Signatur überprüfst.

---

01 Ein Test-Zertifikat erzeugen

Für dieses Kapitel genügt ein selbstsigniertes Zertifikat:

```
openssl req -x509 -newkey rsa:2048 -keyout 304_example_key.pem \
  -out 304_example_cert.pem -days 3650 -nodes \
  -subj "/CN=BinoBI Handbuch Beispiel/O=BinoBI Beispielprojekt/C=DE"
```

`-nodes` erzeugt einen unverschlüsselten privaten Schlüssel – praktisch für dieses Beispiel, in der Produktion solltest du einen passwortgeschützten Schlüssel und `ConnectionSecret`-artige Geheimnisverwaltung (Kapitel 301) in Betracht ziehen.

---

02 Das SigningProfile-Manifest

```
apiVersion: bino.bi/v1alpha1
kind: SigningProfile
metadata:
  name: 304_example_signing_profile
spec:
  certificate:
    path: ../resources/signing/304_example_cert.pem
  privateKey:
    path: ../resources/signing/304_example_key.pem
  digestAlgorithm: sha256
  certType: approval
  docMDPPerm: form-fill-sign
  signer:
    name: "BinoBI Handbuch Beispiel"
    reason: "Kapitel 304 Demonstration"
    location: "Beispielprojekt"
```

Attribut	Pflicht	Beschreibung
`certificate`	ja	PEM-Zertifikat, via `path` (Datei) oder `inline` (String).
`privateKey`	ja	PEM-Private-Key, ebenfalls `path` oder `inline`.
`signer.name`	ja	Name des Unterzeichners, erscheint in der Signatur.
`signer.location` / `reason` / `contact`	nein	Zusätzliche Signer-Metadaten.
`tsaURL`	nein	URL eines Timestamp-Authority-Dienstes für einen vertrauenswürdigen Zeitstempel.
`digestAlgorithm`	nein	`sha256` (Standard), `sha384` oder `sha512`.
`certType`	nein	`certification`, `approval` (Standard), `usage-rights` oder `timestamp`.
`docMDPPerm`	nein	`no-changes`, `form-fill-sign` (Standard) oder `annotate` – erlaubte Änderungen nach dem Signieren.

`certificate` / `privateKey` verwenden **exakt eine** der beiden Quellen `path` (Pfad relativ zur Manifest-Datei) oder `inline` (PEM-Inhalt direkt im Manifest – nur für Testzwecke empfehlenswert, da sonst der private Schlüssel im Klartext im Repository landet).

---

03 Ein ReportArtefact signieren

```
apiVersion: bino.bi/v1alpha1
kind: ReportArtefact
metadata:
  name: bn304_signed_pdf
spec:
  title: Beispiel zur PDF-Signierung
  filename: bn304_signed.pdf
  format: a4
  orientation: landscape
  language: de
  signingProfile: 304_example_signing_profile
  layoutPages:
    - page: bn107_grid
```

`signingProfile` referenziert das `SigningProfile` per `metadata.name`. Dasselbe Feld existiert auch auf `DocumentArtefact` (Kapitel 302).

---

04 Signatur überprüfen

Nach `bino build --artefact bn304_signed_pdf` lässt sich die Signatur mit `pdfsig` (Teil von `poppler-utils`) unabhängig von BinoBI verifizieren:

```
pdfsig dist/bn304_signed.pdf
```

```
Signature #1:
  - Signer Certificate Common Name: BinoBI Handbuch Beispiel
  - Signing Time: Jul 10 2026 20:11:38
  - Signing Hash Algorithm: SHA-256
  - Signature Type: adbe.pkcs7.detached
  - Total document signed
```

```
  - Signature Validation: Signature is Valid.
  - Certificate Validation: Certificate issuer isn't Trusted.
```

“Signature is Valid” bestätigt, dass die Signatur kryptographisch korrekt ist und das Dokument seit der Signierung nicht verändert wurde. **“Certificate issuer isn’t Trusted”** ist bei einem selbstsignierten Test-Zertifikat erwartet – ein produktives Zertifikat einer anerkannten CA würde hier als vertrauenswürdig erkannt.

---

05 Zusammenfassung

1. `SigningProfile` signiert PDFs aus `ReportArtefact` / `DocumentArtefact` digital, referenziert per `spec.signingProfile: <name>`.
2. `certificate` / `privateKey` kommen als PEM, entweder `path` (Datei) oder `inline` (String) – bevorzuge `path` mit einem Secret-Management-Ansatz für echte Zertifikate.
3. `certType` / `docMDPPerm` steuern Zertifikatstyp und erlaubte Nachbearbeitung; `tsaURL` fügt einen vertrauenswürdigen Zeitstempel hinzu.
4. Signaturen sind mit Standardwerkzeugen wie `pdfsig` unabhängig von BinoBI überprüfbar.
5. Ein selbstsigniertes Test-Zertifikat validiert kryptographisch korrekt, wird aber als “Issuer nicht vertrauenswürdig” markiert – für den produktiven Einsatz brauchst du ein Zertifikat einer anerkannten CA.

> **Jetzt bist du dran!** Erzeuge dein eigenes Test-Zertifikat, signiere damit ein `DocumentArtefact` (Kapitel 302) statt eines `ReportArtefact`, und prüfe die Signatur mit `pdfsig`.

305 - Plugin-Engine

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/) sowie im Plugin-Guide unter https://cli.bino.bi/guides/plugins/ (https://cli.bino.bi/guides/plugins/).
>
> **Hinweis:** Dieses Projekt deklariert selbst keine Plugins (`bino plugin list` meldet "No plugins declared in bino.toml."), daher sind die Beispiele in diesem Kapitel illustrativ und nicht gegen ein echtes Plugin getestet.

BinoBI kann über ein Plugin-System um zusätzliche Manifest-Kinds, DataSource-Typen, Lint-Regeln und CLI-Befehle erweitert werden, ohne den `bino` -Kern selbst zu verändern. Plugins sind eigenständige, native Programme, die von `bino` per RPC angesprochen werden.

Du lernst:

- wie ein Plugin in `bino.toml` deklariert und konfiguriert wird,
- wo `bino` Plugin-Binaries sucht,
- was ein Plugin alles beisteuern kann,
- und wie du Plugins auflistest, ausführst und ihre Schemas einsiehst.

---

01 Deklaration in `bino.toml`

Jedes Plugin bekommt einen eigenen `[plugins.<name>]` -Abschnitt:

```
[plugins.salesforce]
path = ".bino/plugins/bino-plugin-salesforce" # optional, sonst Suchpfade (Absch
version = "1.2.0"                             # optionales Versions-Pinning

[plugins.salesforce.config]
api_key = "${SALESFORCE_API_KEY}"
instance_url = "https://example.my.salesforce.com"
```

Die `config` -Map wird dem Plugin beim Start als Schlüssel-Wert-Paare übergeben. **Umgebungsvariablen-Substitution (`${VAR}` , Kapitel 102) funktioniert auch in Config-**

Werten – so hältst du API-Schlüssel aus dem Repository heraus, analog zu den `...FromEnv`-Feldern der `ConnectionSecret`s (Kapitel 301).

---

02 Binaries: Namenskonvention und Suchpfade

Plugin-Binaries müssen dem Namensschema **`bino-plugin-<name>`** folgen (unter Windows mit `.exe`). Ohne explizites `path` sucht `bino` in dieser Reihenfolge:

1. Explizites `path` aus `bino.toml`
2. Projektlokal: `<projekt>/.bino/plugins/`
3. Benutzerweit: `~/.bino/plugins/`
4. System-`PATH`

Da Plugins native Programme sind, brauchst du **pro Plattform ein eigenes Binary** – einen zentralen Plugin-Marktplatz gibt es derzeit nicht, die Verteilung erfolgt manuell (z. B. über GitHub-Releases des Plugin-Anbieters).

---

03 Was ein Plugin beisteuern kann

Fähigkeit	Beschreibung
Eigene DataSources	Neue Manifest-Kinds wie `SalesforceDataSource` (der Ausblick aus Kapitel 103), die in `bino schema` erscheinen.
Visuelle Komponenten	Web Components / eigenes HTML-Rendering für neue Diagramm- oder Anzeigetypen.
Lint-Regeln	Domänenspezifische Prüfungen, die bei `bino lint` mitlaufen.
Pipeline-Hooks	Reaktion auf `post-load`, `post-dataset-execute`, `post-render-html`, `post-render-pdf` – Plugin-seitige Ergänzung zu den Shell-Hooks aus Kapitel 210.
CLI-Befehle	Unterbefehle unter dem Plugin-Namensraum, z. B. `bino plugin exec salesforce:auth`.
Host-Zugriff	Plugins können DuckDB abfragen, DataSet-Ergebnisse lesen und Dokumente des Projekts abrufen.

---

04 Plugins auflisten und ausführen

```
bino plugin list
```

Lädt alle deklarierten Plugins und zeigt Name, Version und Beschreibung. In diesem Projekt:

```
No plugins declared in bino.toml.
```

```
bino plugin exec <plugin>:<command> [args...]
# z. B.:
bino plugin exec example:hello --name World
```

Das Format ist immer `<plugin-name>:<command-name>`, gefolgt von den Argumenten, die der jeweilige Plugin-Befehl erwartet.

---

05 Schema eines Plugin-Kinds einsehen

Bringt ein Plugin eigene Manifest-Kinds mit, erscheinen sie im zusammengeführten Schema. Gezielt abfragen:

```
bino schema --kind SalesforceDataSource
```

Ohne `--kind` gibt `bino schema` das vollständige Schema aller eingebauten **und** Plugin-Kinds aus. Auch die VS-Code-Extension (Kapitel 101) und der MCP-Server (Kapitel 108) nutzen dieses zusammengeführte Schema – Plugin-Kinds bekommen also automatisch Autovervollständigung und Validierung.

---

06 Zusammenfassung

1. Plugins werden pro Plugin in einem **`[plugins.<name>]`**-Abschnitt in `bino.toml` deklariert; `path` und `version` sind optional, `[plugins.<name>.config]` übergibt Konfiguration (inkl. `${VAR}`-Substitution).
2. Binaries heißen **`bino-plugin-<name>`** und werden über `path` → `.bino/plugins/` → `~/.bino/plugins/` → `PATH` gefunden.
3. Plugins können DataSources, visuelle Komponenten, Lint-Regeln, Pipeline-Hooks (`post-load`, `post-dataset-execute`, `post-render-html`, `post-render-pdf`) und CLI-Befehle

beisteuern.

4. **`bino plugin list`** zeigt geladene Plugins, **`bino plugin exec <plugin>:<command>`** führt Plugin-Befehle aus.
5. **`bino schema --kind <name>`** zeigt das Schema eines einzelnen (auch Plugin-)Kinds.

> **Jetzt bist du dran!** Deklariere probeweise einen `[plugins.test]`-Abschnitt mit einem nicht existierenden `path` in `bino.toml` und beobachte, wie `bino plugin list` den Fehlerfall meldet. Entferne den Abschnitt danach wieder.

306 - CI/CD und reproduzierbare Builds

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ (https://cli.bino.bi/reference/).

Dieses letzte Kapitel bündelt die CLI-Funktionen, die für den produktiven Einsatz in einer CI/CD-Pipeline relevant sind: harte Lint-Gates, Abhängigkeits-Inspektion und die Frage, wie reproduzierbar ein `bino build` tatsächlich ist.

Du lernst:

- wie `bino lint --fail-on-warnings` eine Pipeline scheitern lässt,
- wie `bino graph` die Abhängigkeiten eines Projekts sichtbar macht,
- was `SOURCE_DATE_EPOCH` verspricht – und was in einem lokalen Test tatsächlich beobachtet wurde,
- und wann `--data-mode inline` sinnvoll ist.

---

01 Harte Lint-Gates in CI: `--fail-on-warnings`

Standardmäßig gibt `bino lint` Warnungen nur aus, beendet sich aber mit Exit-Code `0`. In einer CI-Pipeline willst du meist, dass **jede** Warnung den Build stoppt:

```
bino lint --execute-queries --fail-on-warnings
```

In diesem Projekt bestehen aktuell mehrere `page-layout-slots-used`-Warnungen (siehe `pages/page_06_test.yaml`). Entsprechend beendet sich der Befehl mit einem von `0` verschiedenen Exit-Code:

```
bino:
✗ Runtime Error

  lint found 7 warning(s)
```

Für ein CI-Gate reicht es, den Exit-Code auszuwerten:

```
bino lint --execute-queries --fail-on-warnings || exit 1
```

01.01 Welche Regeln laufen?

`bino lint` prüft aktuell **16** eingebaute Regeln (plus Schema-Validierung und ggf. Plugin-Regeln, Kapitel 305). Die Regeln melden Warnungen; als **Fehler** wird lediglich `missing-required-reference` gemeldet, das beim Prüfen der Layout-Slots entsteht:

Regel-ID	Schwere	Prüft
`missing-required-reference`	**Error**	Eine `ref`-Referenz zeigt auf kein existierendes Dokument.
`report-artefact-required`	Warning	Mindestens ein `ReportArtefact` ist definiert.
`artefact-layoutpage-required`	Warning	Jedes `ReportArtefact` hat eine passende `LayoutPage`.
`text-content-required`	Warning	`Text`-Komponenten haben ein nicht-leeres `spec.value`.
`dataset-required`	Warning	Visualisierungs-Komponenten haben eine `dataset`-Bindung.
`table-sum-title-unused`	Warning	`sumTitle` steht nicht auf einer Tabelle ohne Summenzeile.
`page-layout-slots-used`	Warning	Kinder-Anzahl passt zur Slot-Anzahl des `pageLayout`.
`card-layout-slots-used`	Warning	Wie oben, für `cardLayout`.
`ref-params`	Warning	An eine `ref` übergebene `params` sind dort auch deklariert.
`asset-reference-undefined`	Warning	Jede `asset:`-Referenz zeigt auf ein existierendes `Asset`.
`asset-source-missing`	Warning	Die Datei hinter einem `Asset` existiert.
`i18n-code-unused`	Warning	Jeder `Internationalization.code` passt zur Sprache eines Artefakts.

Regel-ID	Schwere	Prüft
`i18n-namespace-unreferenced`	Warning	Jeder i18n-Namensraum wird von mindestens einer Komponente genutzt.
`i18n-title-namespace-deprecated`	Warning	`titleNamespace` ist veraltet; `i18nNamespace` verwenden.
`inline-ref-bounds`	Warning	`@inline(N)` -Indizes liegen innerhalb der `dependencies`.
`dataset-source-exclusive`	Warning	`source` wird nicht mit `query` / `prql` kombiniert.
`inline-naming-conflict`	Warning	Kein Dokumentname beginnt mit dem reservierten Präfix `_inline_`.

Die vollständige, aktuelle Liste inkl. Details steht im Anhang der Referenz (cli.bino.bi/appendix/lint-rules/ (https://cli.bino.bi/appendix/lint-rules/)).

Zusätzlich schreiben `bino build` und `bino lint` zeitgestempelte Logdateien nach `dist/*.log` – in einer Pipeline lohnt es sich, diese als Build-Artefakte zu archivieren, um fehlgeschlagene Läufe nachträglich analysieren zu können.

---

02 Abhängigkeiten inspizieren: `bino graph`

```
bino graph --artefact <name> --view tree   # Standard: Baum-Ansicht
bino graph --artefact <name> --view flat   # Tabellen-Ansicht
```

`bino graph` lädt alle Manifeste, baut den DataSource/DataSet/Komponenten/Artefakt-Graphen auf und zeigt für jeden Knoten einen kurzen **Hash** (Fingerabdruck des Manifest-Inhalts). Ein Auszug für `bn107_grid_pdf`:

```
bn107_grid_pdf [ReportArtefact] hash=5f07ef3f96d8, file=reports/handbuch_example
├── ... (alle Komponenten des Projekts, unabhängig vom --artefact-Filter)
└── bn107_grid [LayoutPage] hash=d484790500aa, file=pages/examples/bn107_grid.ya
    └── Grid bn107_grid#0 [Component] hash=851977bec533, kind=Grid, ...
        ├── ChartStructure bn107_grid#0.1 [Component] hash=d040a6de91e5, ...
        │   └── bb_example_dataset [DataSet] hash=ea2360b4819e, ...
        │       └── bb_example_data [DataSource] hash=c9f3fef0e5d2, ...
```

Nützlich, um in einer PR zu prüfen, **welche** Manifest-Hashes sich durch eine Änderung tatsächlich verschoben haben – `bino build` schreibt dieselben Hashes übrigens automatisch in eine `.bngraph`-Datei neben jedem Artefakt (abschaltbar mit `bino build --no-graph`).

> **Beobachtung:** `--artefact` filtert die Baum-Wurzel auf das gewünschte Artefakt, listet in der Baum-Ansicht aber weiterhin **alle** eigenständigen Komponenten-Dateien des Projekts mit auf – nicht nur die tatsächlich vom gewählten Artefakt referenzierten. Verlass dich für "was hängt wirklich von X ab" auf den eingerückten Teilbaum unterhalb des Artefakt-Knotens.

---

03 Reproduzierbare Builds: `SOURCE_DATE_EPOCH`

`bino build --help` erwähnt `SOURCE_DATE_EPOCH` als Umgebungsvariable für "byte-reproduzierbare" PDFs – sie normalisiert eingebettete Zeitstempel und Dokument-IDs auf einen festen Wert, statt die aktuelle Systemzeit zu verwenden:

```
export SOURCE_DATE_EPOCH=1735689600 # fester Unix-Timestamp
bino build --artefact bn107_grid_pdf
```

> **Beobachtung:** In einem lokalen Test unterschied sich der SHA-256-Hash zweier direkt aufeinanderfolgender Builds derselben Seite **mit identisch gesetztem** `SOURCE_DATE_EPOCH` trotzdem. `SOURCE_DATE_EPOCH` normalisiert offenbar nicht alle Quellen für Build-zu-Build-Abweichungen (denkbar sind z. B. intern generierte SVG-Element-IDs). Verlass dich für Integritätsprüfungen deshalb nicht blind auf Byte-Identität – vergleiche stattdessen die `bino graph`-Hashes (Abschnitt 02) der zugrunde liegenden Manifeste, oder den Textinhalt des PDFs, wenn dir Reproduzierbarkeit wichtig ist. Prüfe das Verhalten in jedem Fall mit deiner eigenen `bino`-Version, bevor du dich in einer Pipeline darauf verlässt.

---

04 `--data-mode`: eingebettete vs. verlinkte Daten

```
bino build --artefact <name> --data-mode url     # Standard: Daten per URL/Refer
bino build --artefact <name> --data-mode inline  # Daten gzip+base64 im HTML ein
```

Zugehörige Feinsteuerung: `--embed-data-base64` (Standard `true`), `--embed-data-csv`, `--embed-data-max-bytes` (Standard 65536), `--embed-data-max-rows` (Standard 10), `--embed-

`data-redact` (Standard `true`, maskiert eingebettete Werte). `--data-mode inline` ist z. B. nützlich, wenn ein erzeugtes PDF vollständig eigenständig sein soll (etwa für ein Archiv), ohne dass zugehörige Datendateien separat mitgeliefert werden müssen.

---

05 Zusammenfassung

1. **`bino lint --fail-on-warnings`** macht Warnungen zu einem harten CI-Gate (Exit-Code ≠ 0 bei Warnungen).
2. **`bino graph --view tree|flat`** zeigt den DataSource/DataSet/Komponenten/Artefakt-Graphen inkl. Inhalts-Hashes – praktisch für Code-Reviews.
3. **`SOURCE_DATE_EPOCH`** soll Builds reproduzierbar machen, hat das im Test aber nicht vollständig erreicht – prüfe das eigenständig, bevor du dich darauf verlässt.
4. **`--data-mode inline`** (mit `--embed-data-*`-Flags) bettet Beispieldaten direkt ins PDF ein, statt sie nur zu verlinken.

> **Jetzt bist du dran!** Baue eine kleine CI-Pipeline-Definition (z. B. GitHub Actions) mit drei Schritten: `bino lint --execute-queries --fail-on-warnings`, `bino build`, und einem Schritt, der `bino graph --view flat` in die Build-Logs schreibt.

307 – XY-Charts im Detail: Token, Ebenen und Skalierung

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/chart-scatter/ (https://cli.bino.bi/reference/chart-scatter/).

Kapitel 211 und 212 haben `ChartScatter`, `ChartBubble` und `ChartBullet` als Werkzeuge vorgestellt. Dieses Kapitel geht eine Ebene tiefer: Wie genau werden Messwerte adressiert, wie hängen die drei Ebenenbegriffe zusammen, warum ist Skalierung bei Flächen ein anderes Problem als bei Balken – und was passiert davon eigentlich wo.

Du lernst:

- die vollständige Grammatik der Measure-Token inklusive Varianzen,
- warum alle Achsen einer Szenario-Familie angehören müssen,
- wie `level`, `seriesLevel` und `facet` eine Hierarchie bilden,
- wie `ScalingGroup` Flächen über Charts hinweg vergleichbar macht,
- warum `scale` bei Blasen quadratisch wirkt,
- wie sich `limit` zwischen Scatter und Bubble unterscheidet,
- und welche Komponente welche RuleSet-Eigenschaft tatsächlich ausliest.

---

01 Die Grammatik der Measure-Token

Die Mappings `x`, `y`, `size` und `share` erwarten einen **Token** – entweder direkt oder als `measure` innerhalb eines Objekts.

01.01 Szenario-Slots

Die Grundform sind die Spalten des Standard-Datenschemas:

```
ac1 … ac4      Ist (actual)
pp1 … pp4      Vorperiode (previous period)
fc1 … fc4      Forecast
pl1 … pl4      Plan
```

01.02 Varianz-Token

Auf den **Achsen** von `ChartScatter` und `ChartBubble` sind zusätzlich Varianzen erlaubt:

Form	Bedeutung	Beispiel
`d<basis>_<delta>`	absolute Abweichung	`dac1_pp1` – Ist minus Vorjahr
`dr<basis>_<delta>`	relative Abweichung (in Prozent)	`drac1_pl1` – Planabweichung in %

Ein angehängtes `_pos`, `_neg` oder `_neu` wird **akzeptiert und auf Achsen ignoriert** – diese Sentiment-Suffixe steuern die Einfärbung in Tabellen, für eine Achsenposition sind sie bedeutungslos. `drac1_pl1_pos` und `drac1_pl1` verhalten sich auf einer Achse also identisch.

> **Wichtig:** Diese Token adressieren **Spalten, die im DataSet existieren müssen**. BinoBI rechnet sie nicht selbst aus. Die Abweichung gehört ins SQL:
>
> ```
> SELECT category, categoryIndex, operation, ac1, pp1, pl1,
>        round((ac1 - pp1) * 100.0 / pp1, 1) AS drac1_pp1,
>        round((ac1 - pl1) * 100.0 / pl1, 1) AS drac1_pl1
> FROM einheiten_raw
> ```

Auf `ChartBullet` sind Varianz-Token dagegen **nicht** zulässig: Die Abweichung ist dort implizit der Vergleich von `actual` und `target`.

01.03 Die Regel der Szenario-Familie

> Alle **einfachen** Szenario-Messwerte eines Charts müssen derselben Familie angehören.

`x: ac1` zusammen mit `y: ac2` ist zulässig. `x: ac1` mit `y: pp2` nicht – das Diagramm könnte die Marken nicht eindeutig behandeln, weil die Familie über das RuleSet die Darstellung wählt.

Varianz-Token zählen nicht zur Familie. Ein Chart mit `x: drac1_pp1` und `y: drac1_pl1` ist deshalb völlig in Ordnung – genau das ist das typische Varianz-Portfolio.

---

02 Ebenen: `level`, `seriesLevel` und `facet`

Drei Attribute greifen auf dieselbe Hierarchie zu (`rowGroup` → `category` → `subCategory`) und müssen in einer festen Ordnung zueinander stehen:

```
facet.level      (Panel)     ── am weitesten oben
   └── seriesLevel  (Farbe)
          └── level      (Punkt)  ── am weitesten unten
```

Attribut	Antwortet auf	Werte
`level`	Was ist **ein Punkt**?	`auto` , `category` , `subcategory`
`seriesLevel`	Was bestimmt die **Farbe**?	`auto` , `rowgroup` , `category` , `none`
`facet.level`	Was erzeugt ein **eigenes Panel**?	`rowgroup` , `category`

`seriesLevel` muss **echt oberhalb** von `level` liegen. Identifiziert `category` den Punkt, kann die Farbe nach `rowGroup` gruppieren – nicht umgekehrt. `seriesLevel: none` zeichnet einfarbig.

02.01 Die Entdopplung – und warum sie stillschweigend passiert

Der Punkt-Identität liegt der **Name** auf der `level` -Ebene zugrunde. Tritt derselbe Name mehrfach auf, behält BinoBI die erste Zeile und verwirft die übrigen.

Das ist im Alltag die häufigste Ursache für „im Chart fehlen Punkte":

```
# Falsch: "Cloud" kommt in EMEA und APAC vor -> APAC verschwindet
- { rowGroup: "EMEA", category: "Cloud", ... }
- { rowGroup: "APAC", category: "Cloud", ... }

# Richtig: eindeutige Namen auf der Punkt-Ebene
- { rowGroup: "EMEA", category: "Cloud EMEA", ... }
- { rowGroup: "APAC", category: "Cloud APAC", ... }
```

> **Technischer Hintergrund:** Die Entdopplung findet in der Template-Engine beim Rendern statt, nicht beim Laden der Manifeste. `bino lint` und `bino build` melden deshalb **nichts** – die Warnung landet in der Browser-Konsole. Wenn ein XY-Chart zu wenige Punkte zeigt, ist das die erste Hypothese, die du prüfen solltest, nicht die letzte.

02.02 Small Multiples

`facet` zerlegt die Zeichenfläche in Panels – ein bewährtes Mittel, wenn eine Serie sonst zu dicht wird:

```
spec:
  x: { measure: drac1_pp1, label: "Wachstum", unit: "%", refLine: 0 }
  y: { measure: drac1_pl1, label: "Planabweichung", unit: "%", refLine: 0 }
  seriesLevel: rowgroup
  facet:
    level: rowgroup
    columns: 2
  labels:
    points: all
```

Mit `refLine: 0` auf beiden Achsen entstehen vier Quadranten mit klarer Bedeutung: rechts oben wächst *und* übertrifft den Plan, links unten weder noch.

▶ **Zum Ausprobieren:** Varianz-Portfolio mit Facets in der Sandbox öffnen (https://sandbox.bino.bi/s/g7f1lwbc9l2w5o9) – die Abweichungsspalten werden im DataSet per SQL berechnet. Setze `columns` auf `1` und beobachte, wie sich das Seitenverhältnis der Panels verändert.

> Quelle im Repo: 307_01_scatter_varianz_facet.yaml

---

03 `limit`: zwei Komponenten, zwei Bedeutungen

Ein Detail, das leicht übersehen wird, weil das Attribut gleich heißt:

Komponente	`limit` behält …
`ChartScatter`	die **ersten Zeilen in DataSet-Reihenfolge**
`ChartBubble`	die **größten Blasen nach** `size`
`ChartBullet`	die ersten Zeilen – **ohne** Restzeile

Bei `ChartScatter` bestimmt also dein `ORDER BY` im DataSet, was überlebt. Bei `ChartBubble` entscheidet die Größe – was fast immer das ist, was man will, aber eben nicht von der Sortierung abhängt.

Bei `ChartBullet` wird bewusst **keine „Sonstige"-Zeile** gebildet: Die Summe aus Umsatz, Marge und Kopfzahl wäre bedeutungslos.

---

04 `ScalingGroup`: Flächen vergleichbar machen

Standardmäßig normalisiert jedes Blasendiagramm **für sich**: Der größte Wert bekommt den maximalen Radius. Zwei Charts nebeneinander sind damit *nicht* vergleichbar – eine Blase gleicher Größe kann links 320 und rechts 180 bedeuten.

`ScalingGroup` behebt das durch eine feste Zuordnung von Wert zu Fläche:

```
apiVersion: bino.bi/v1alpha1
kind: ScalingGroup
metadata:
  name: umsatz_flaeche
spec:
  value: 12
---
apiVersion: bino.bi/v1alpha1
kind: ChartBubble
metadata:
  name: portfolio_emea
spec:
  dataset: einheiten_emea
  x: { measure: ac1, label: "Marktanteil", min: 0, max: 3 }
  y: { measure: ac2, label: "Wachstum", unit: "%", min: 0, max: 20 }
  size: { measure: ac3, label: "Umsatz", unit: "mEUR", group: umsatz_flaeche }
```

04.01 Map-Scale-Semantik

Alle `ScalingGroup`-Werte folgen derselben Logik – **Datenmenge pro Darstellungseinheit**. Ein *größerer* Wert bedeutet also *kleinere* Marken:

Konsument	Einheit von `value`
`unitScaling` (Tabelle, Charts)	Dateneinheiten je **1 em Balkenlänge**
`percentageScaling`	Prozentpunkte je **1 em Pin-Länge**
`size.group` (`ChartBubble`)	Dateneinheiten je **1 em² Blasenfläche**

Weil der Bezug em-basiert ist, funktioniert dieselbe Gruppe für Tabellen und Charts gleichermaßen – jede Komponente rechnet relativ zu ihrer eigenen Schriftgröße.

> **Für echte Vergleichbarkeit reicht `size.group` allein nicht.** Fixiere zusätzlich identische `min` / `max` auf beiden Achsen. Sonst sind die Flächen zwar vergleichbar, die Positionen

> aber nicht.

Ein unbekannter Gruppenname ist ein **blockierender Render-Fehler**, kein stiller Fallback.

► **Zum Ausprobieren:** Synchronisierte Blasenflächen in der Sandbox öffnen (https://sandbox.bino.bi/s/zm3fn7tq1ny2jqw) – zwei Charts, eine `ScalingGroup`. Entferne `group: umsatz_flaeche` aus einem der beiden und vergleiche: Die Blasen sehen plötzlich ähnlich groß aus, obwohl EMEA fast doppelt so viel Umsatz trägt.

> Quelle im Repo: 307_02_scalinggroup_bubbles.yaml

04.02 Warum `scale` bei Blasen quadratisch wirkt

`scale` skaliert die effektive Schriftgröße einer Komponente. Da Blasenflächen über `ScalingGroup` in **em²** definiert sind, wirkt ein Faktor auf die Fläche im **Quadrat**:

```
scale: 1.5   ->   Fläche × 2,25
scale: 0.5   ->   Fläche × 0,25
```

Bei Balken (Länge in em) ist der Effekt linear. Wer zwei Blasencharts mit unterschiedlichem `scale` nebeneinanderstellt, hat die Vergleichbarkeit trotz gemeinsamer `ScalingGroup` wieder verloren – setze in diesem Fall auf beiden denselben Faktor oder `scale: none`.

---

05 Iso-Linien: Voraussetzungen

`iso` zeichnet Hyperbeln `x · y = k`. Damit das definiert ist, müssen **beide Achsen einen echt positiven Wertebereich** haben. Enthält eine Achse Null oder negative Werte, ist die Iso-Linie in diesem Bereich nicht darstellbar.

In der Praxis heißt das: Für Varianz-Portfolios (Achsen um 0 herum) sind Iso-Linien das falsche Werkzeug. Für Marge-gegen-Umsatz sind sie ideal, weil das Produkt eine reale Größe ist – den Rohertrag.

`values: auto` überlässt BinoBI die Wahl sinnvoller Stufen; eine explizite Liste ist vorzuziehen, wenn die Schwellen fachlich gesetzt sind.

---

06 Welche Komponente liest welche RuleSet-Eigenschaft?

Ein RuleSet (Kapitel 213) stellt drei Eigenschaften bereit, aber nicht jede Komponente nutzt jede. Der folgende Stand wurde an der Template-Engine `v1.0.0-next.22` überprüft:

Eigenschaft	Ausgewertet von
`sortIndex`	`Table`, `ChartStructure`, `ChartTime`, Titelzeile
`colorIndex`	`ChartScatter`, `ChartBubble`, `ChartBullet`
`name`	derzeit von **keiner** Komponente für sichtbare Beschriftungen

Daraus folgen zwei praktische Konsequenzen:

- **`colorIndex` in einem RuleSet wirkt auf die XY-Charts**, nicht auf Tabellen oder Struktur-Diagramme – die leiten ihre Behandlung aus der Szenario-Familie ab.
- **`sortIndex` wirkt umgekehrt** auf Tabellen, Struktur- und Zeitdiagramme, nicht auf XY-Charts – dort gibt es keine Spaltenreihenfolge, die zu sortieren wäre.

Die Details zu `name` und zur Herkunft der sichtbaren Kürzel stehen in Kapitel 308, Abschnitt 05.

---

07 Seitenverhältnis und Layout

XY-Charts haben – anders als `ChartStructure` – **keine datengetriebene Höhe**. Zehn Punkte brauchen genauso viel Platz wie hundert. Die Zeichenfläche wird deshalb über `aspect` als Verhältnis `"w:h"` bestimmt (Standard `16:9`).

Daraus ergibt sich eine Layout-Regel: In einer schmalen Karte ist `4:3` oder `1:1` meist lesbarer, in einem breiten Banner `21:9`. Bei `facet` gilt `aspect` je **Panel**, nicht für die Gesamtfläche – bei vier Panels in zwei Spalten wird die Komponente also etwa doppelt so hoch wie ein einzelnes Panel.

---

08 Checkliste für XY-Charts

- ☐ Ist im DataSet **eine Zeile je Punkt** – also fertig aggregiert?
- ☐ Sind die Namen auf der `level`-Ebene **eindeutig**?
- ☐ Gehören alle einfachen Szenario-Messwerte **einer Familie** an?
- ☐ Existieren die Varianz-Spalten **als Spalten** im DataSet?
- ☐ Sind `min` / `max` fixiert, wenn zwei Charts verglichen werden?
- ☐ Ist bei `compareWith` die Vergleichsfamilie in **allen** Slots vorhanden?
- ☐ Liegen `share`-Werte zwischen **0 und 1**?

- [ ] Ist die Template-Engine mindestens `v1.0.0-next.19`?

---

09 Tipps

- **Varianzen ins SQL, nicht ins Chart.** Das macht sie in `bino lint --execute-queries` prüfbar und in der Datenvorschau sichtbar.
- `facet` **vor** `filter`**:** Vier Panels sagen mehr als vier separate Charts mit je einem Filter – und bleiben automatisch synchron skaliert.
- `ScalingGroup` **gehört ins Projekt, nicht in die Komponente.** Ein zentral gepflegter Wert ist der einzige Weg, Vergleichbarkeit dauerhaft zu halten.
- **Bei leerem Chart zuerst die Engine-Version prüfen** – das kostet zehn Sekunden und ist die häufigste Ursache.

> Vollständige Referenz: cli.bino.bi/reference/chart-scatter/ (https://cli.bino.bi/reference/chart-scatter/), chart-bubble (https://cli.bino.bi/reference/chart-bubble/), chart-bullet (https://cli.bino.bi/reference/chart-bullet/), scaling-group (https://cli.bino.bi/reference/scaling-group/)

308 – Das Szenario-Modell: RuleSets im Detail

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/reference/ruleset/ (https://cli.bino.bi/reference/ruleset/).

Kapitel 213 hat gezeigt, *wie* man ein RuleSet schreibt. Dieses Kapitel erklärt, *was dabei tatsächlich passiert*: wie die Ebenen zusammengeführt werden, wie ein Präfix zu vier Varianten expandiert, welche Komponente welche Eigenschaft ausliest – und warum eine sichtbare Umbenennung von Szenarien derzeit nicht über das RuleSet läuft.

Die Angaben in diesem Kapitel wurden gegen die Template-Engine `v1.0.0-next.22` geprüft.

Du lernst:

- wie der Drei-Ebenen-Merge funktioniert und was „partiell" konkret bedeutet,
- wie aus einem Präfix vier Varianten mit abgeleiteten Sortierwerten werden,
- wie Varianz-Präfixe ihren Sortierbereich erhalten,
- welche Komponente welche Eigenschaft liest,
- woher die sichtbaren Szenario-Kürzel wirklich kommen,
- und wie du RuleSets systematisch debuggst.

---

01 Die drei Ebenen

Jede Komponente führt bei jedem Rendern drei Ebenen zusammen. Später gewinnt:

```
1. _system    eingebaute IBCS-Vorgaben (nicht änderbar)
       ↓
2. _default   dein globales RuleSet, falls vorhanden
       ↓
3. <benannt>  das über spec.ruleset referenzierte RuleSet
```

Der Merge ist **tief und partiell**. Jede Ebene überschreibt ausschließlich die Felder, die sie tatsächlich nennt; alles Übrige fällt auf die darunterliegende Ebene zurück. Ein RuleSet ist damit immer ein **Diff**, kein vollständiges Regelwerk.

01.01 Der Merge in der Praxis

Das folgende Bundle macht die Ebenen direkt sichtbar – zwei Tabellen auf derselben Seite, dieselben Daten, ein Unterschied:

```
# Ebene 2: gilt automatisch für alle Komponenten
apiVersion: bino.bi/v1alpha1
kind: RuleSet
metadata:
  name: _default
spec:
  content:
    scenarios:
      fc:
        sortIndex: 350        # Forecast direkt vor das Ist
---
# Ebene 3: nur für Komponenten, die es referenzieren
apiVersion: bino.bi/v1alpha1
kind: RuleSet
metadata:
  name: planHinten
spec:
  content:
    scenarios:
      pl:
        sortIndex: 900
```

Das Ergebnis:

Tabelle	Wirksame Ebenen	Spaltenreihenfolge
ohne `ruleset`	`_system` + `_default`	`PY | PL | FC | AC`
mit `ruleset: planHinten`	`_system` + `_default` + `planHinten`	`PY | FC | AC | PL`

Beachte die zweite Zeile genau: `planHinten` erwähnt **nur** `pl`. Der Forecast steht trotzdem weiterhin vor dem Ist – dieser Wert stammt unverändert aus `_default`. Genau das ist mit „partiell" gemeint.

► **Zum Ausprobieren:** Merge-Ebenen in der Sandbox öffnen (https://sandbox.bino.bi/s/5qa1zfjlk51jb6r) – beide Tabellen nebeneinander. Forke das Beispiel und lösche das `_default`-Dokument: Beide Tabellen verlieren die Forecast-Umsortierung, die untere behält aber die Plan-Regel.

> Quelle im Repo: 308_01_ruleset_merge.yaml

02 Von Präfix zu Variante

Ein RuleSet-Eintrag beschreibt ein **Präfix** (`ac`), nicht einen konkreten Slot (`ac1`). Beim Aufbau der internen Szenario-Tabelle wird jedes Präfix auf vier Varianten expandiert:

Ableitung	Regel	Beispiel bei `ac` (`sortIndex: 400`)
`name`	Variantennummer wird angehängt	`AC` → `AC1` , `AC2` , `AC3` , `AC4`
`sortIndex`	Basis + `n * 10`	410, 420, 430, 440
`colorIndex`	unverändert von der Basis übernommen	10 für alle vier

Daraus folgt eine Regel für eigene Sortierwerte: **Lass zwischen zwei Präfixen mindestens 40 Punkte Abstand**, sonst greifen die Varianten ineinander. Die eingebauten Werte halten mit ihren 100er-Schritten reichlich Abstand – bei eigenen Werten wie `sortIndex: 405` für ein zweites Präfix wäre die Reihenfolge dagegen von der Variantennummer abhängig und praktisch nicht mehr vorhersagbar.

Deshalb ist `sortIndex: 900` in den Beispielen kein Zufall: Der Wert liegt weit genug jenseits von `ac` (400 → 440), dass alle vier Plan-Varianten sicher dahinter landen.

---

03 Varianz-Präfixe

Neben den Basis-Szenarien kennt das Modell zwei abgeleitete Präfix-Familien:

Familie	Präfixe	`sortIndex`
Basis	`ac` , `fc` , `pl` , `bu` , `pp` , `py`	wie definiert
absolute Varianz	`dac` … `dpy`	Basis + 1000
relative Varianz	`drac` … `drpy`	Basis + 2000

Der Aufbau sorgt dafür, dass Abweichungsspalten immer **hinter** allen Szenariospalten stehen und relative hinter absoluten – ohne dass du dich darum kümmern musst.

Wichtig: Diese Werte werden aus der jeweiligen **Basis** abgeleitet. Verschiebst du `pl` auf 900, wandert `dpl` auf 1900 und `drpl` auf 2900 mit. Die relative Ordnung bleibt also erhalten – ein Grund mehr, Präfixe zu verschieben statt einzelne Varianten.

Eigene, nicht vorgesehene Präfixe sind erlaubt; sie fallen ohne eigenen Eintrag auf die `fallback` -Regel zurück.

04 Wer liest was?

Nicht jede Komponente wertet jede Eigenschaft aus. Der folgende Stand wurde an der Engine `v1.0.0-next.22` überprüft:

Eigenschaft	Ausgewertet von	Wirkung
`sortIndex`	`Table`, `ChartStructure`, `ChartTime`, Titelzeile	Spalten- und Serienreihenfolge
`colorIndex`	`ChartScatter`, `ChartBubble`, `ChartBullet`	IBCS-Flächenbehandlung
`name`	–	derzeit ohne sichtbare Wirkung

Das erklärt zwei Beobachtungen, die zunächst widersprüchlich wirken:

- Ein RuleSet mit `colorIndex` ändert an einer **Tabelle** nichts. Tabellen und Struktur-Diagramme leiten ihre Darstellung aus der Szenario-Familie ab, nicht aus dem `colorIndex`.
- Ein RuleSet mit `sortIndex` ändert an einem **XY-Chart** nichts. Dort gibt es keine Spaltenreihenfolge.

Wenn ein RuleSet „nicht wirkt", ist die erste Frage deshalb nicht, ob es geladen wurde – sondern ob die Zielkomponente die geänderte Eigenschaft überhaupt liest.

---

05 Woher die sichtbaren Kürzel kommen

Die Referenz beschreibt `name` als Anzeigename des Szenarios. In der aktuell ausgelieferten Engine trifft das für die sichtbaren Beschriftungen **nicht** zu – ein `name: PLAN` ändert die Spaltenüberschrift einer Tabelle nicht.

Die tatsächlich gerenderten Kürzel stammen aus dem **Internationalisierungs-Store**, der die Szenario-Slots als globale Label-Keys führt:

```
ac1 → "AC"      pp1 → "PY"      fc1 → "FC"      pl1 → "PL"
ac2 → "AC2"     pp2 → "PY2"     …
```

Daran erkennt man den Mechanismus auch von außen: Die eingebauten RuleSet-Vorgaben nennen für `pp` den Namen `PP`, gerendert wird aber `PY`. Käme die Beschriftung aus dem RuleSet, stünde dort `PP`.

Was das für die Praxis heißt:

- Verlass dich für Umbenennungen **nicht** auf `name`.
- `sortIndex` und `colorIndex` wirken zuverlässig – dafür sind RuleSets das richtige Werkzeug.
- Prüfe eine geplante Umbenennung immer an einem gebauten PDF, nicht am Manifest.
- Für abweichende Beschriftungen ist die Internationalisierung (Kapitel 208) der naheliegende Ansatzpunkt; die Slot-Keys werden dort allerdings als globale Keys behandelt und lassen sich nicht in jedem Fall überschreiben.

> Diese Einschränkung ist versionsabhängig. Prüfe bei einem Engine-Update, ob sich das Verhalten geändert hat – ein kurzer Build mit einem Test-RuleSet genügt.

---

06 RuleSets debuggen

Wenn eine Regel nicht greift, arbeite diese Reihenfolge ab:

1. **Wird das Manifest überhaupt geladen?** `bino lint` listet alle Dokumente; ein RuleSet in einem per Constraint ausgeschlossenen Bereich taucht dort nicht auf.
2. **Liest die Zielkomponente die Eigenschaft?** Siehe die Tabelle in Abschnitt 04 – das ist die häufigste Ursache.
3. **Gibt es eine explizite Liste?** Eine Tabelle mit `scenarios: [...]` legt ihre Spalten selbst fest; `sortIndex` ist dann wirkungslos.
4. **Greift die Vererbung?** `inherited-page` funktioniert nur, wenn die umgebende `LayoutPage` tatsächlich ein `ruleset` gesetzt hat – sonst bleibt es beim `_default`.
5. **Kollidieren Sortierwerte?** Prüfe den Abstand zwischen den Präfixen (Abschnitt 02).
6. **Vergleiche gegen einen Referenzlauf.** Zwei Komponenten mit denselben Daten nebeneinander, eine mit und eine ohne `ruleset` – so wie in Abschnitt 01. Das trennt „Regel wirkt nicht" zuverlässig von „Regel wirkt anders als erwartet".

---

07 Muster für die Praxis

07.01 Hausweite Konvention

Ein einziges `_default`-RuleSet im Projekt, das die Abweichung von IBCS dokumentiert:

```
apiVersion: bino.bi/v1alpha1
kind: RuleSet
metadata:
  name: _default
  description: "Hausstandard: Forecast direkt vor dem Ist"
spec:
  content:
    scenarios:
      fc:
        sortIndex: 350
```

Die `description` ist hier kein Beiwerk – sie ist der Ort, an dem die *Begründung* für eine Abweichung vom Standard steht.

07.02 Ausnahme für eine einzelne Seite

Benanntes RuleSet plus `inherited-page` auf den Kindern – so gilt die Ausnahme für die ganze Seite inklusive Titelzeile, ohne dass jede Komponente sie einzeln nennt (Kapitel 213, Abschnitt 05).

07.03 Umgebungsabhängige Regeln

RuleSets sind normale Manifeste und akzeptieren `metadata.constraints`. Ein RuleSet lässt sich damit auf ein Format oder einen Modus einschränken:

```
apiVersion: bino.bi/v1alpha1
kind: RuleSet
metadata:
  name: _default
  constraints:
    - "format==a4"
spec:
  content:
    scenarios:
      fc:
        sortIndex: 350
```

Achte dabei darauf, dass für jede Kombination **höchstens ein** `_default` aktiv ist – zwei gleichzeitig geladene Dokumente gleichen Namens lassen die Duplikatsprüfung fehlschlagen.

---

08 Tipps

- **RuleSets sind Diffs.** Je kürzer, desto verständlicher – und desto weniger bricht bei einem Engine-Update.
- **Abstände von 100** zwischen eigenen Sortierwerten halten Varianten sauber getrennt und lassen Platz für Späteres.
- **Präfixe verschieben, nicht Varianten.** Die abgeleiteten Varianz-Familien folgen dann automatisch.
- **Jede Abweichung von IBCS begründen** – in `metadata.description`. Der nächste Mensch am Projekt liest sie.
- **Nach jedem Engine-Update kurz gegenprüfen**, ob die RuleSet-Eigenschaften noch dieselbe Wirkung haben. Die Zuordnung in Abschnitt 04 ist versionsabhängig.

> Vollständige Referenz: cli.bino.bi/reference/ruleset/ (https://cli.bino.bi/reference/ruleset/)

309 – bino im Container: reproduzierbare Builds mit Docker

> **Disclaimer:** Die in diesem Kapitel vorgestellten Attribute sind nicht immer vollständig. Die vollständige und aktuelle Liste findest du in der Referenz unter https://cli.bino.bi/guides/docker/ (https://cli.bino.bi/guides/docker/).

Kapitel 306 hat gezeigt, wie ein Build in einer CI-Pipeline läuft – inklusive `bino setup`, um Chrome und die Template-Engine nachzuladen. Das funktioniert, hat aber zwei Schwächen: Der Download kostet bei jedem Lauf Zeit, und „die neueste Version" ist keine reproduzierbare Angabe.

Das offizielle Container-Image löst beides. Es bringt alles mit, was zum Rendern nötig ist, lädt zur Laufzeit **nichts** nach und verhält sich auf dem Laptop, auf dem CI-Runner und im abgeschotteten Cluster gleich.

Du lernst:

- welche zwei Image-Varianten es gibt und wann welche passt,
- wie du dein Projekt korrekt in den Container mountest,
- wie du Secrets und Umgebungsvariablen hineinreichst,
- warum `bino preview` und `bino serve` im Container ein `--addr` brauchen,
- wie du Dateirechte unter Linux in den Griff bekommst,
- was offline funktioniert und was nicht,
- und wie die Engine-Version im Image mit deiner `bino.toml` zusammenspielt.

---

01 Die zwei Varianten

Variante	Image	Größe	Inhalt
Full	`ghcr.io/bino-bi/bino-cli:latest`	~1,1 GB	Alles, inklusive Chromium und Schriften
Slim	`ghcr.io/bino-bi/bino-cli:latest-slim`	~410 MB	Alles **außer** Chromium

Die Slim-Variante kann `serve`, `lint`, `graph`, `lsp` und `mcp`. Sie kann **nicht** `build` und `preview` – beide brauchen einen Browser zum Rendern.

Die Aufteilung folgt einem klaren Zweck: Ein `bino serve`-Deployment rendert nie ein PDF und braucht darum keinen Browser. Für einen Build-Runner ist das Full-Image richtig.

Eigenschaft	Wert
Registry	`ghcr.io/bino-bi/bino-cli`
Tags	`latest`, `X.Y`, `vX.Y.Z` – jeweils auch als `-slim`
Architekturen	`linux/amd64`, `linux/arm64`
Basis	Debian Bookworm (glibc)
Entrypoint	`bino`
Arbeitsverzeichnis	`/work`
User	uid `1000`, group `0`
`HOME`	`/opt/bino`
Exponierter Port	`8080`

---

02 Der erste Lauf

```
docker pull ghcr.io/bino-bi/bino-cli:latest
docker run --rm ghcr.io/bino-bi/bino-cli:latest version
```

Der Entrypoint **ist** bereits `bino`. Alles nach dem Image-Namen ist ein Unterkommando:

```
docker run --rm ghcr.io/bino-bi/bino-cli:latest --help
```

> **Häufiger Fehler:** `docker run ... IMAGE bino build` schlägt mit `executable file not found` fehl – der Binary-Name wird doppelt angegeben. Richtig ist `docker run ... IMAGE build`.

> **Immutable Tags verwenden.** `latest` wandert mit jedem Release. Ein Bericht, der heute rendert, soll nächstes Quartal identisch rendern – dafür gehört ein `vX.Y.Z`-Tag in die

> Pipeline. `latest` ist für Wegwerf-Experimente.

---

03 Das Projekt mounten

Gemountet wird die **Bundle-Wurzel** – das Verzeichnis mit der `bino.toml` – nach `/work`:

```
docker run --rm -v "$PWD:/work" ghcr.io/bino-bi/bino-cli:latest lint
docker run --rm -v "$PWD:/work" ghcr.io/bino-bi/bino-cli:latest build --out-dir
```

`/work` ist ein Bind-Mount, keine Kopie – die PDFs landen also direkt in `dist/` deines echten Projektverzeichnisses.

> **`/work` muss beschreibbar sein.** `bino build` legt `dist/` und `.bino/cache/` innerhalb des Bundles an; ein Read-only-Mount schlägt fehl.

03.01 Secrets und Variablen hineinreichen

`-e VAR` **ohne Wert** reicht die Variable aus deiner Shell durch – nichts Sensibles landet damit in der Shell-History:

```
docker run --rm \
  -v "$PWD:/work" \
  -e DB_HOST \
  -e POSTGRES_PASSWORD \
  ghcr.io/bino-bi/bino-cli:latest \
  build --out-dir dist/
```

Diese Variablen erreichen sowohl die `${VAR}`-Ersetzung in den Manifesten als auch die `*FromEnv`-Felder eines `ConnectionSecret` (Kapitel 301).

> Fehlt eine Variable, meldet der Build `unresolved variable ${...}`. Im Container ist die Ursache fast immer ein vergessenes `-e` – nicht ein Fehler im Manifest.

---

04 Dateirechte

Der Container läuft als uid `1000`. Was das für die erzeugten Dateien bedeutet, hängt vom Host ab:

Plattform	Verhalten
macOS, Windows (Docker Desktop)	Die Dateifreigabe bildet die Rechte auf deinen User ab – nichts zu tun.
Linux	Dateien gehören uid `1000`. Auf Einzelplatzrechnern bist das meist du selbst.

Unter Linux mit abweichender uid überschreibst du den User – die Caches im Image sind für Gruppe `0` lesbar, jede uid funktioniert also, solange die Gruppe bleibt:

```
docker run --rm \
  -u "$(id -u):0" \
  -v "$PWD:/work" \
  ghcr.io/bino-bi/bino-cli:latest build --out-dir dist/
```

> **`HOME` dabei nicht überschreiben.** Die vorinstallierten DuckDB-Erweiterungen und die Template-Engine liegen unter `/opt/bino/.bino`. Mit einem anderen `HOME` findet bino sie nicht, versucht sie zur Laufzeit nachzuladen – und scheitert ohne Netzwerkzugang. Genau der Vorteil des Images wäre damit verspielt.

04.01 Ein Shell-Wrapper

Die volle `docker run`-Zeile tippt niemand gern zweimal. Eine Funktion in `~/.zshrc` oder `~/.bashrc` genügt:

```
bino() {
  docker run --rm -it \
    -v "$PWD:/work" \
    -p 127.0.0.1:8080:8080 \
    ghcr.io/bino-bi/bino-cli:latest "$@"
}
```

Danach verhalten sich `bino lint`, `bino build` und `bino version` wie eine lokale Installation.

> `bino init` und `bino add` sind interaktiv – behalte dafür `-it`. In CI lässt du `-t` weg (es gibt kein TTY) und ergänzt `-y` für den nicht-interaktiven Lauf.

---

05 Preview und Serve im Container

Hier lauert die Stolperfalle, die die meiste Zeit kostet:

> `bino preview` und `bino serve` binden standardmäßig an `127.0.0.1`. Im Container ist das die **Loopback-Adresse des Containers** – ein veröffentlichter Port zeigt dann ins Leere.

Die Lösung ist `--addr`:

```
docker run --rm -it -v "$PWD:/work" -p 127.0.0.1:8080:8080 \
  ghcr.io/bino-bi/bino-cli:latest preview --addr 0.0.0.0:8080
```

Danach erreichst du die Vorschau unter `http://localhost:8080/` – auf Linux, macOS und Windows gleichermaßen, ohne `--network host`.

> **Sicherheitshinweis:** `--addr 0.0.0.0:8080` öffnet den Preview-Server für alles, was den veröffentlichten Port erreicht – einschließlich `POST /__explorer/query`, das **beliebiges DuckDB-SQL** gegen dein Projekt ausführt, ohne Authentifizierung. Veröffentliche den Port darum an Loopback (`-p 127.0.0.1:8080:8080`), solange du das Netz nicht kontrollierst. Das obige Beispiel tut genau das.

Für das tägliche Arbeiten bleibt eine lokal installierte `bino` das angenehmere Werkzeug. Das Image spielt seine Stärken bei reproduzierbaren Builds und beim Ausliefern von `bino serve` aus.

---

06 Ein neues Projekt anlegen

`bino init` legt sein Zielverzeichnis **relativ zum Arbeitsverzeichnis** an – gemountet wird deshalb das *übergeordnete* Verzeichnis:

```
docker run --rm -it -v "$PWD:/work" ghcr.io/bino-bi/bino-cli:latest \
  init standard -d monatsbericht --title "Monatsbericht" --language de -y
```

Die eingebauten Vorlagen `minimal` und `standard` stecken in der Binary und brauchen kein Netz. Entfernte Vorlagen (`bino init owner/repo`) dagegen schon.

---

07 Was offline funktioniert

Fest im Image enthalten und ohne jeden Netzwerkzugriff nutzbar:

- die `bino`-Binary,
- Chromium und ein Schriftsatz für den PDF-Export (Full-Variante) – ein `bino setup` entfällt,
- die DuckDB-Erweiterungen `excel`, `httpfs`, `postgres`, `mysql`, `prql` und `webdavfs`,
- das Bundle der Template-Engine.

Das lässt sich verifizieren, indem du dem Container das Netz vollständig entziehst:

```
docker run --rm --network none -v "$PWD:/work" \
  ghcr.io/bino-bi/bino-cli:latest build --out-dir dist/
```

Diese Funktionen brauchen weiterhin Netz:

Funktion	Grund
Entfernte Projektvorlagen (`bino init owner/repo`)	Werden aus dem Quell-Repository geladen.
`bino registry`	Spricht mit der Paket-Registry (Kapitel 214).
Präsentationen	Die Reveal.js-Laufzeit kommt beim Betrachten von einem CDN.
DataSources auf HTTP, S3 oder Datenbanken	Sie erreichen *deine* Daten.
`bino update`	Hier nicht verwenden – stattdessen den Image-Tag ändern.

> **Präsentationen sind die eine Ausnahme**, die sich nicht offline betreiben lässt: Ohne ausgehende Verbindung bleibt das Foliendeck leer. Die PDF-Ausgabe desselben `ReportArtefact` ist davon nicht betroffen (Kapitel 102).

---

08 Die Engine-Version im Image

Jedes Image trägt **genau eine** Template-Engine-Version, hinterlegt als Image-Label:

```
docker image inspect ghcr.io/bino-bi/bino-cli:latest \
  --format '{{ index .Config.Labels "bi.bino.engine-version" }}'
```

> **Kritisch:** Pinne in `bino.toml` **keine** `engine-version`, die von der im Image abweicht – und niemals `engine-version = "SNAPSHOT"`. Für eine gepinnte Version fällt bino **nicht** auf einen Download zurück; der Build bricht mit einem Missing-Engine-Fehler ab. Entweder du lässt den Pin weg und nutzt die mitgelieferte Engine, oder du pinnst exakt die Version aus dem Image.

Das ist besonders relevant, wenn du die XY-Charts aus Kapitel 211 und 212 einsetzt: Sie brauchen mindestens `v1.0.0-next.19`. Prüfe das Label, bevor du einen Image-Tag in der Pipeline festschreibst.

---

09 Fehlerdiagnose

Symptom	Ursache	Lösung
`bino: executable file not found`	Binary-Name doppelt angegeben	`docker run ... IMAGE build`, nicht `... IMAGE bino build`
bino meldet ein leeres Projekt	Nichts oder das falsche Verzeichnis nach `/work` gemountet	Aus der Bundle-Wurzel starten, `-v "$PWD:/work"`
`"google-chrome": executable file not found`	Slim-Variante ohne Browser	Full-Image für `build` und `preview` verwenden
Dateien gehören uid 1000 (Linux)	Container-User weicht ab	`-u "$(id -u):0"` ergänzen, `HOME` unangetastet lassen
Browser erreicht Preview/Serve nicht	Bindung an die Container-Loopback	`--addr 0.0.0.0:8080` übergeben
`template engine ... not found`	`engine-version` passt nicht zum Image	Pin entfernen oder exakt angleichen
Folien bleiben leer oder ungestylt	Reveal.js kommt vom CDN	Egress zu `cdn.jsdelivr.net` erlauben oder als PDF ausliefern

Symptom	Ursache	Lösung
`unresolved variable ${...}`	Variable nicht in den Container gereicht	`-e VAR` oder `--env-file` ergänzen
Datenbankverbindung abgelehnt	`localhost` ist im Container der Container selbst	`host.docker.internal`, den Compose-Servicenamen oder den echten Hostnamen verwenden

Der letzte Punkt kostet erfahrungsgemäß am meisten Zeit: Eine `ConnectionSecret`-Konfiguration, die lokal funktioniert, zeigt im Container auf den Container.

---

10 Tipps

- **Version pinnen, nicht `latest`.** Der ganze Sinn des Images ist Reproduzierbarkeit.
- **Slim für `serve`, Full für `build`.** 700 MB Ersparnis pro Deployment sind ein Argument.
- **`--network none` als Test in der Pipeline:** Er beweist, dass der Build wirklich hermetisch ist – und deckt vergessene Remote-Abhängigkeiten auf, bevor sie im abgeschotteten Netz auffallen.
- **Engine-Label und `bino.toml` gemeinsam prüfen**, wenn ein Diagramm im Container leer bleibt, lokal aber rendert.
- **Den Wrapper an Loopback binden.** `-p 8080:8080` ohne Adresse veröffentlicht auf allen Interfaces – inklusive des SQL-Endpunkts der Vorschau.

> Vollständige Referenz: cli.bino.bi/guides/docker/ (https://cli.bino.bi/guides/docker/)