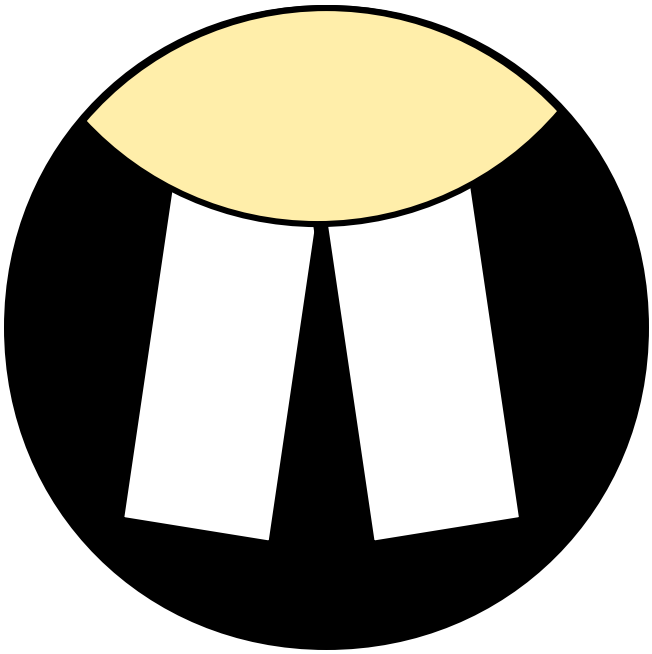

Pfarrplaner
Technisches Handbuch

Architektur, Entwicklung, Deployment und API-Dokumentation

Christoph Fischer

Pfarrplaner v.2026.12.0, Stand: 27.05.2026

Inhaltsverzeichnis

Technisches Handbuch

Dieses Handbuch beschreibt die technische Seite von Pfarrplaner: Architektur, Verzeichnisstruktur, Laufzeit, Deployment und API.

Schnellzugriff

Dieses Handbuch ist richtig für Sie, wenn ...

- Sie Pfarrplaner technisch betreiben oder weiterentwickeln.
- Sie verstehen möchten, wo welche Logik liegt.
- Sie mit der API arbeiten oder sie erweitern möchten.

Andere Handbücher

Für tägliche Bedienung gibt es das Benutzerhandbuch.

Für Installation, Wartung und Updates gibt es das Administratorhandbuch.

1. Einführung

Pfarrplaner ist eine Laravel-Anwendung mit Inertia.js und Vue. Das System verbindet klassische Serverlogik, ein modernes Single-Page-Frontend, dokumentenorientierte Ausgaben und eine gewachsene API-Landschaft für Kalender-, Liturgie- und Verwaltungsfunktionen.

1.1. Technischer Kern

- Backend: Laravel
- Frontend: Inertia.js mit Vue
- Build: Vite
- Authentifizierung: Laravel-Mechanismen und API-Schutz über `auth:api` bzw. Sanctum-Bausteine im Projekt
- Dokumente und Exporte: PDF-, Office- und HTML-Ausgaben
- Browser-Automatisierung: Puppeteer und Browsershot

1.2. Wichtige Betriebsmodi

- klassische PHP-Webanwendung hinter Apache oder Nginx
- optional erweiterte Laufzeit mit Octane/RoadRunner
- lokale Entwicklungsumgebungen
- Docker-Compose-Umgebungen

1.3. Drei technische Sichtweisen

1. **Domänenlogik**: Gottesdienste, Kasualien, Benutzer, Orte, Urlaub, Liturgie.
2. **Betrieb**: `.env`, Datenbank, Storage, Queue, Scheduler, Logs, Chromium.
3. **Schnittstellen**: Web-Routen, API-Routen, OpenAPI, Exporte, externe Integrationen.

2. Architektur und Domänen

2.1. Backend-Struktur

Das Projekt folgt im Kern einer Laravel-Struktur, nutzt aber mehrere projektweite Konventionen:

- Modelle liegen domänenorientiert unter `app/Models/`
- Web- und API-Controller sind getrennt
- Geschäftslogik wandert häufig in `app/Actions/`
- Modelle registrieren ihre Routen selbst über `registerRoutes()` und `registerApiRoutes()`

2.2. Zentrale Domänen

- **Gottesdienste und Veranstaltungen**: `Service`, Kalender, Mitwirkende, Werbung, Berichte
- **Kasualien**: Taufen, Trauungen, Beerdigungen
- **Benutzer und Rechte**: Personen, Rollen, Gemeindezuordnungen
- **Orte und Gemeinden**: Kirchengemeinden, Pfarrämter, Veranstaltungsorte, Sitzpläne
- **Urlaub und Vertretung**: Abwesenheiten, Pools, Poolmaster
- **Liturgie**: Bausteine, Lieder, Psalmen, liturgische Texte

2.3. Routing-Modell

- Web-Routen werden zentral geladen und um zusätzliche Teilrouten ergänzt.
- API-Routen liegen unter `routes/api/*.php`.
- Modelle können eigene Web- und API-Routen registrieren.

2.4. Frontend-Modell

Das Frontend nutzt Inertia. Viele Seiten werden also serverseitig als Inertia-Responses erzeugt, während Vue die eigentliche Interaktion im Browser übernimmt.

Der Help-Button in der Topbar wird serverseitig mit einer Manual-Zuordnung versorgt und kann dadurch jetzt zwischen Benutzer- und Administratorhandbuch unterscheiden.

2.5. Dokumentenerzeugung

Pfarrplaner erzeugt nicht nur HTML, sondern viele Ausgabeformate:

- PDF
- Word
- Excel
- HTML-Snippets
- Browser-basierte Renderings

Darum gehören Office-Bibliotheken, PDF-Rendering und Chromium-Laufzeit fest zur technischen Gesamtarchitektur.

3. Dateien und Verzeichnisse

3.1. Besonders wichtige Verzeichnisse

- `app/` : Anwendungslogik
- `app/Http/Controllers/Api/` : API-Controller
- `app/Actions/` : fachliche Aktionen
- `app/Models/` : Domänenmodelle
- `config/` : Laufzeitkonfiguration
- `database/` : Migrationen, Factories, Seeder
- `manual/` : Benutzerhandbuch
- `manual/administratorhandbuch/` : Administratorhandbuch
- `manual/technikhandbuch/` : Technisches Handbuch
- `resources/js/` : Vue- und Inertia-Frontend
- `routes/` : Web-, API- und Teilrouten
- `scripts/` : Build-, Release- und Hilfsskripte
- `storage/` : Logs, temporäre Dateien, Anwendungsdaten

3.2. Dokumentationsrelevante Dateien

- `config/manual.php` : Zuordnung von Anwendungsrouten zu Handbuchkapiteln
- `scripts/build-manual-site.js` : baut alle drei Manuals als statische Site und PDF
- `scripts/deploy-manual-site.js` : deployt die gebaute Handbook-Site
- `app/Console/Commands/DevBuilder/BuildManualPages.php` : erzeugt Benutzerhandbuch-Seiten wie Versionsangaben und Lizenzen

3.3. Installations- und Update-relevante Dateien

- `app/Console/Commands/InstallCommand.php` : neuer CLI-Installer `pfarrplaner:install`
- `app/Console/Commands/Install/InstallUpdates.php` : Laravel-Einstiegspunkt für Updates
- `scripts/install-updates.js` : Update-Ablauf

3.4. API-relevante Dateien

- `routes/api.php` : lädt alle API-Teilrouten
- `routes/api/*.php` : Route-Definitionen
- `app/Console/Commands/GenerateOpenApiSpec.php` : erzeugt `openapi.json`
- `app/Http/Controllers/Api/OpenApiSpec.php` : zentrale OpenAPI-Tags und Serverdefinition

4. Laufzeit, Build und Hintergrundprozesse

4.1. PHP-Laufzeit

Pfarrplaner setzt derzeit PHP `^8.4` voraus. Die Anwendung läuft klassisch als Laravel-Webanwendung und kann zusätzlich mit Octane-/RoadRunner-Bausteinen erweitert werden.

4.2. Frontend-Build

Das Frontend wird mit Vite gebaut. Relevante Schritte:

- `npm install`
- `npm run build`
- lokal optional `npm run dev`

4.3. Dokumentations-Build

Für die Handbook-Site gibt es eine eigene Pipeline:

- `php artisan build:manual`
- `node scripts/build-manual-site.js`
- optional `node scripts/deploy-manual-site.js`

Der Handbook-Build erzeugt jetzt:

- `benutzerhandbuch/`
- `administratorhandbuch/`
- `technikhandbuch/`
- eine gemeinsame Landing-Page unter `https://handbuch.pfarrplaner.de/`

4.4. Screenshot-Workflow

Benutzerhandbuch-Screenshots werden über Dusk-Tests unter `tests/Browser/Manual/` erzeugt. Der kombinierte Build mit temporärem Laravel-Server läuft über `scripts/build-manual-with-server.js`.

4.5. Browser-Tests (Dusk)

Für lokale Browser-Tests steht `npm run test:dusk` zur Verfügung. Der Aufruf startet `php artisan dusk:run --compact` mit der kompakten Collision-Ausgabe im Stil von `php artisan test --compact`. Während längerer stiller Phasen gibt der Runner zusätzlich regelmäßige Statusmeldungen aus, damit ein langsamer Browser-Test nicht wie ein Hänger aussieht.

Zusätzlich gibt es gezielte Varianten für die lokale Fehlersuche:

- `npm run test:dusk:compact` verwendet ebenfalls die kompakte Collision-Ausgabe und erlaubt zusätzliche Optionen wie `--stop-on-failure`.
- `npm run test:dusk:debug` zeigt die jeweils laufenden PHPUnit-Browser-Tests direkt im Terminal an.
- `npm run test:dusk:testdox` verwendet die PHPUnit-TestDox-Ausgabe.

Der Dusk-Command baut das Frontend standardmäßig genau einmal selbst, setzt die SQLite-Testdatenbank zurück und startet bei Bedarf einen lokalen Laravel-Testserver. Für schnellere Wiederholungsläufe stehen unter anderem diese Optionen zur Verfügung:

- `php artisan dusk:run --without-build` verwendet den vorhandenen Vite-Build erneut.
- `php artisan dusk:run --stop-on-failure` bricht nach dem ersten fehlschlagenden Browser-Test ab.
- `php artisan dusk:run --without-server` verwendet einen bereits laufenden lokalen Server mit passender `APP_URL`.

Entwickler-Hilfsrouten werden in Pfarrplaner bewusst restriktiv behandelt:

- `APP_DEBUG` fällt ohne explizite Umgebungsvariable auf `false` zurück.
- Laravel Boost ist standardmäßig deaktiviert und muss über `BOOST_ENABLED=true` bewusst eingeschaltet werden.
- Ignition-Housekeeping und runnable solutions bleiben standardmäßig deaktiviert und werden nur bei expliziter Freigabe über Umgebungsvariablen aktiviert.
- Telescope bleibt standardmäßig deaktiviert und wird nur über `TELESCOPE_ENABLED=true` bewusst freigeschaltet.
- Dusk-Hilfsrouten werden zusätzlich durch `DUSK_ENABLED=true` abgesichert und sind nur für gezielte Browser-Testläufe vorgesehen.

4.6. Scheduler und Queue

Technisch gehören diese Prozesse fest zur Laufzeit:

- `schedule:run`
- Queue-Worker
- Neustart der Worker nach Code- oder Konfigurationswechsel

4.7. Browser-Laufzeit

Da Pfarrplaner Puppeteer und Browsershot nutzt, ist ein funktionierender Chromium-Stack Teil der Laufzeit. Fehler in diesem Bereich zeigen sich häufig erst bei PDF-, Screenshot- oder Exportfunktionen.

5. Deployment und Betrieb

5.1. Klassisches Deployment

Der klassische Weg kombiniert:

- Git-Checkout
- Composer
- npm/Vite
- `.env`
- Migrationen
- Cache- und Prozesspflege

Für Erstinstallationen ist `php artisan pfarrplaner:install` der vorgesehene Weg. Für Updates dient `php artisan install:updates`.

5.2. Docker-Deployment

Im Containerbetrieb sollte das Deployment reproduzierbar über Images und Compose-Dateien laufen. Entscheidend sind persistente Volumes, konsistente `.env`-Werte und ein sauberer Umgang mit Migrationen und Worker-Prozessen.

5.3. Release-Kontext

Das Repository enthält Release-Skripte, die neben Versionsständen auch den Neubau und die Auslieferung des Handbuchs berücksichtigen.

5.4. Health und Monitoring

Für Monitoring ist mindestens der Health-Endpunkt sinnvoll:

- `GET /api/health`

Zusätzlich sollten Sie Webserver, PHP-Prozesse, Queue, Scheduler und Datenbank überwachen.

5. Sicherheitsaudit

Dieses Kapitel beschreibt den Stand eines vollständigen Sicherheitsdurchgangs auf Web-, API- und Download-Oberflächen von Pfarrplaner.

5.1. Ziel des Audits

Geprüft wurden insbesondere:

- ungeschützte Web- und API-Routen
- benutzerdefinierte Controller-Aktionen ohne Policy- oder Gate-Prüfung
- Aktionen, die verschachtelte Ressourcen ohne Besitzprüfung verändern
- Selbstbedienungsfunktionen rund um Profil, Tokens und Benutzerwechsel
- öffentliche Endpunkte mit möglicher Datenpreisgabe
- alte Debug-, Entwickler- und Wartungsoberflächen

5.2. Ergebnisübersicht

Der produktive Routenbestand wurde geprüft und gezielt gehärtet. Dabei wurden folgende Klassen von Problemen beseitigt:

- Debug- und Diagnose-Routen wurden entfernt oder standardmäßig blockiert.
- Nicht verwendete DGM-Anmeldung wurde vollständig entfernt.
- Öffentliche Schreibzugriffe in Kasual-, Streaming- und Dienstpfaden wurden mit Authentifizierung und Policies abgesichert.
- Mehrere Controller erhielten fehlende `Gate::authorize(...)` -Prüfungen.
- Verschachtelte Routen für Blöcke, Liturgie-Items und Anhänge prüfen jetzt die Zugehörigkeit zum Elternobjekt.
- Selbstbedienungsfunktionen für Profil, Einstellungen, API-Tokens und Benutzerwechsel wurden nachgeschärft.
- Zustandsändernde Alt-Routen wurden von `GET` auf `POST` umgestellt.
- Öffentliche Datei- und Bildzugriffe auf `attachments/*` benötigen nun Anmeldung oder eine signierte URL.
- Die öffentliche Urlaubs-Einbettung eines beliebigen Benutzers ist nicht mehr anonym erreichbar.

5.3. Sicherheitsmodell nach dem Audit

5.3.1. Routenmatrix nach Schutzklasse

Die Routen wurden nicht nur stichprobenartig, sondern nach Schutzklassen durchgegangen. Für den laufenden Betrieb ist die Einteilung wie folgt relevant:

Schutzklasse	Typische Routenfamilien	Ergebnis
authentifiziert	Verwaltung, Bearbeitung, Admin, Profil, Tokens, Berichte, Inputs, Liturgie-Editor, Gottesdienste, Kasualien, API, Extranet, DAV	auf Anmeldung begrenzt; kritische Sonderaktionen zusätzlich mit Objektprüfung nachgeschärft
signiert oder authentifiziert	`image/{path}`, `files/{path}`, signierte Liedblatt-Downloads, signierte öffentliche Anfrage-Links	anonym nur mit gültiger Signatur, intern weiterhin nach Anmeldung nutzbar
bewusst öffentlich lesend	öffentliche Informationsseiten, Podcasts, Kalender- und Einbettungsansichten, QR-Codes, Kinderkirchen-Ansichten, öffentliche Liturgie	keine allgemeinen Schreibrechte; öffentliche Embed- und Download-Pfade gezielt geprüft
bewusst öffentlich schreibend	`dimissoriale/{type}/{id}`, `anfrage/{ministry}/{user}/{services}/{sender?}`, `kontaktformular`, technische Sonderfälle wie `csrf-cookie`	nur verbleibend, wenn fachlich erforderlich; signierte oder fachlich eingegrenzte Verarbeitung
paketbedingt registriert, operativ blockiert	`_dusk/*`, `_ignition/*`	nicht als normale Produktoberfläche nutzbar; standardmäßig blockiert oder praktisch stillgelegt

5.3.2. Öffentlich bewusst erreichbar

Diese Endpunkte bleiben öffentlich, weil sie Produktfunktion sind:

- ausgewählte öffentliche Inhaltsseiten wie `was-ist-der-pfarrplaner`
- öffentliche Liturgieansicht `mitfeiern/{service:slug}`
- öffentliche Einbettungen für Gottesdienst-, Kinderkirchen- und Streaming-Ausgaben
- Podcast-Feed
- QR-Code-Ausgabe
- signierte öffentliche Workflows wie Ministry-Request, Dimissoriale und signierte Liedblatt-Downloads

5.3.3. Öffentlich, aber technisch eingeschränkt

- Laravel-Dusk-Routen können paketbedingt registriert sein, werden aber serverseitig blockiert, solange `DUSK_ENABLED` nicht ausdrücklich gesetzt ist.
- Ignition-Routen können paketbedingt registriert sein, sind aber ohne aktivierte Runnable-Solutions praktisch stillgelegt.
- Datei- und Bildrouten für `attachments/*` akzeptieren anonym nur noch signierte URLs.

5.3.4. Authentifiziert erforderlich

Die Verwaltungs-, Bearbeitungs- und API-Oberflächen für Benutzer, Dienste, Kasualien, Liturgie, Urlaub, Teams, Orte, Rollen und Konfiguration sind auf Authentifizierung begrenzt und wurden in den kritischen Sonderaktionen zusätzlich gegen fehlende Objektberechtigung geprüft.

5.4. Wichtige behobene Schwachstellen

5.4.1. 1. Öffentliche Mutationen

Mehrere spezielle Aktionen waren zwar fachlich intern gedacht, aber nicht ausreichend mit Authentifizierung oder Policy-Prüfungen abgesichert.

Behoben in:

- Dienst-, Streaming- und Kasual-Controller
- öffentliche Dimissorial- und Ministry-Request-Flows

5.4.2. 2. Objektverwechslung in verschachtelten Routen

Bei Routen wie `service -> block -> item` oder `rite -> attachment` konnten zuvor fremde Kindobjekte über direkte IDs adressiert werden.

Jetzt wird das Kindobjekt jeweils über die Elternbeziehung aufgelöst oder gegen diese geprüft.

5.4.3. 3. Selbstbedienung und Rechteausweitung

Nachgeschärft wurden:

- Benutzer-Impersonation
- Passwort-Reset
- Stadtberechtigungen bei Benutzeränderungen
- Einstellungen und Token-Erzeugung

5.4.4. 4. Öffentliche Dateipfade

Die generischen Endpunkte `image/{path}` und `files/{path}` waren für `attachments/*` anonym nutzbar. Dadurch waren erratbare oder bekannte Dateinamen öffentlich abrufbar.

Jetzt gilt:

- intern: Zugriff nach Anmeldung
- öffentlich: Zugriff nur mit signierter URL

5.5. Regressionstests

Für den Audit wurden spezielle Sicherheitsprüfungen als Tests ergänzt:

- `tests/Builder/SecurityRouteAuditTest.php`
- `tests/Feature/SecurityPublicAssetAccessFeatureTest.php`

Diese Tests prüfen unter anderem:

- entfernte Debug-Routen
- keine sensiblen Zustandsänderungen mehr per `GET`
- keine unerwarteten öffentlichen Mutationsrouten
- Schutz öffentlicher Datei- und Bildpfade

5.6. Verbleibende bewusst öffentliche Schreibpfade

Nach Abschluss des Audits bleiben nur noch fachlich oder technisch begründete öffentliche Schreibpfade übrig:

- `POST dimissoriale/{type}/{id}` : nur mit gültiger signierter Anforderung sinnvoll nutzbar
- `POST anfrage/{ministry}/{user}/{services}/{sender?}` : Diensteauswahl durch signierte Vorgabeliste begrenzt
- `POST kontaktformular` : öffentlicher Kontaktkanal ohne Objektmutation
- `POST _ignition/execute-solution` und `POST _ignition/update-config` : paketbedingt vorhanden, aber ohne aktivierte Runnable-Solutions praktisch stillgelegt
- `GET|POST csrf-cookie` : technischer Bootstrap-Endpunkt

5.7. Restliche Aufgaben nach diesem Audit

Der kritische Härtungsdurchgang ist abgeschlossen. Weitere Prüfungen bleiben normale Weiterentwicklungsarbeit:

- neue Spezialrouten bei künftigen Features wieder gegen diese Checkliste prüfen
- bei neuen öffentlichen Ausgaben immer zwischen offen, signiert und authentifiziert unterscheiden
- bei neuen verschachtelten Routen die Eltern-Kind-Zugehörigkeit explizit absichern

6. API-Authentifizierung

Die API ist standardmäßig vollständig mit `auth:api` abgesichert. Öffentliche API-Routen sind nur noch dort zulässig, wo die Öffentlichkeit fachlich eindeutig beabsichtigt ist.

6.1. Was das praktisch bedeutet

- Grundsätzlich braucht jeder `/api` -Endpunkt eine Anmeldung.
- Alle schreibenden API-Aufrufe sind geschützt.
- Auch lesende Endpunkte gelten standardmäßig als intern oder benutzerbezogen und sind deshalb ebenfalls geschützt.
- Aktuell ist nur der Health-Check unter `/api/health` öffentlich erreichbar.

6.2. Projektbezug

Im Projekt sind unter anderem diese Bausteine sichtbar:

- Guard `api` aus `config/auth.php` mit Token-Authentifizierung
- Middleware `auth:api` als Standard für die komplette API
- Web-Authentifizierung für normale Oberflächenrouten
- interne Vue-Komponenten verwenden für API-Aufrufe bevorzugt `this.$api()` , damit das Benutzer-Token konsistent als Bearer-Token gesendet wird

6.3. Empfehlung für Integrationen

- Neue produktive Integrationen dürfen nicht von implizit öffentlichen Endpunkten ausgehen.
- Integrationen sollten immer ein Benutzer-API-Token mitsenden und nicht auf Cookie-Sitzungen vertrauen.
- Vor dem Einsatz sind Route, Controller, erwartete Rechte und Datenumfang zu prüfen.
- Für dokumentierte Schnittstellen sollte zusätzlich die generierte OpenAPI-Spezifikation bereitgestellt werden.

6.4. Typische geschützte Bereiche

- Kalenderdetails und Schnellauswahlen
- Gottesdienst-CRUD und Personenzuordnungen
- Liturgie-Bearbeitung, Lied- und Textsuche
- Teams, Gemeinden, Pools, Berichte und Inbox-Funktionen

6.5. Typische öffentliche oder halböffentliche Bereiche

- Health-Check unter `/api/health`

Wenn ein neuer API-Endpunkt öffentlich bleiben soll, muss das in Route, Controller und Dokumentation ausdrücklich erkennbar sein. Ein „historisch gewachsener“ öffentlicher GET-Endpunkt ist kein ausreichender Grund.

7. API-Endpunkte

Dieses Kapitel beschreibt die aktuell sichtbaren API-Routen aus `routes/api/*.php`. Für neue Integrationen sollten Sie zusätzlich die Controller und nach Möglichkeit die generierte OpenAPI-Spezifikation prüfen.

7.1. Allgemeine Hinweise

- Basispräfix: `/api`
- Antwortformat ist in der Regel JSON.
- Alle `/api`-Endpunkte sind standardmäßig mit `auth:api` geschützt.
- Die einzige dokumentierte öffentliche Ausnahme ist derzeit `/api/health`.
- Nicht jede Route ist bereits vollständig mit OpenAPI-Attributen beschrieben.

7.2. Health

Methode	Pfad	Schutz	Zweck
GET	/api/health	öffentlich	einfacher Health-Check für Anwendung und Datenbank

7.3. Kalender und Veranstaltungen

Methode	Pfad	Schutz	Zweck
GET	/api/cal/city/{city}/{date}	auth:api	Kalenderdaten für Gemeinde und Monat
GET	/api/quick-pick/{date}	auth:api	Schnellansicht für ein Datum
GET	/api/kalender/monat/{date}	auth:api	Monatsansicht
GET	/api/kalender/gottesdienst/{service}	auth:api	Details zu einem Gottesdienst im Kalenderkontext
GET	/api/veranstaltungen/{calendar}/{start}/{end}	auth:api	Veranstaltungen in einem Bereich

7.4. Gottesdienste

Methode	Pfad	Schutz	Zweck
GET	/api/servicesByDayAndCity/{day}/{city}	auth:api	Gottesdienste eines Tages und einer Gemeinde
GET	/api/user/{user}/services	auth:api	Gottesdienste einer Person
GET	/api/service/{service}	auth:api	Detailansicht eines Gottesdiensts
PATCH	/api/service/{service}	auth:api	Gottesdienst aktualisieren
DELETE	/api/service/{service:slug}	auth:api	Gottesdienst löschen
POST	/api/service/{service}/assign	auth:api	Personen oder Dienste zuordnen
GET	/api/gottesdienste/{date}/{cities}	auth:api	Monatsabfrage für mehrere Gemeinden

7.5. Liturgie und Texte

Methode	Pfad	Schutz	Zweck
POST	/api/liturgy/tree/save/{service}	auth:api	Sortierung des Liturgiebaums speichern
POST	/api/liturgy/service/{service}/import/{source}	auth:api	Liturgie aus Quelle importieren
POST	/api/liturgy/service/{service}/blocks	auth:api	Liturgieblock anlegen
PATCH	/api/liturgy/block/{block}	auth:api	Liturgieblock ändern
DELETE	/api/liturgy/block/{block}	auth:api	Liturgieblock löschen
POST	/api/liturgy/block/{block}/items	auth:api	Liturgieelement anlegen
PATCH	/api/liturgy/item/{item}	auth:api	Liturgieelement ändern
DELETE	/api/liturgy/item/{item}	auth:api	Liturgieelement löschen
POST	/api/liturgy/item/{item}/assign	auth:api	Verantwortliche für Element setzen
GET	/api/liturgy/texts/list	auth:api	Liste liturgischer Texte
POST	/api/liturgy/text/import	auth:api	Text aus Word importieren
GET	/api/liturgy/sources/{serviceId}	auth:api	Importquellen für Liturgie
GET	/api/texte/liste	auth:api	Liste liturgischer Texte

7.6. Lieder und Gesangbücher

Methode	Pfad	Schutz	Zweck
GET	/api/liturgy/songs	auth:api	Liedliste
GET	/api/liturgy/songselect	auth:api	reduzierte Liedauswahl
GET	/api/liturgy/songs/songbooks	auth:api	zugehörige Gesangbücher
PATCH	/api/liturgy/songs/{song}	auth:api	Lied ändern
GET	/api/liturgie/lied/{song}/noten/{verses?}/{lineNumber?}	auth:api	Noten-/Musikdarstellung
GET	/api/liturgy/song/{songReferenceId}	auth:api	einzelnes Lied
POST	/api/liturgy/songs	auth:api	Lied anlegen
GET	/api/songbooks/colors	auth:api	Farbzuordnung für Gesangbücher

7.7. Personen, Teams und Gemeinden

Methode	Pfad	Schutz	Zweck
GET	/api/people/select	auth:api	Personenauswahl
POST	/api/people/search/{searchString}	auth:api	Personensuche
POST	/api/people/activate/{user}/{city}	auth:api	Person für Gemeinde aktivieren
GET	/api/teams/by-city/{city}	auth:api	Teams einer Gemeinde
GET	/api/city/{city}/konfiapp-types	auth:api	KonfiApp-Typen einer Gemeinde

7.8. Urlaub, Pools und Vertretung

Methode	Pfad	Schutz	Zweck
POST	/api/absence/{absence}/set-checked	auth:api	Abwesenheit als geprüft markieren
POST	/api/absence/{absence}/set-approved	auth:api	Abwesenheit genehmigen
DELETE	/api/absence/{absence}	auth:api	Abwesenheit löschen
GET	/api/pools/{pool}/poolmasters/{date}	auth:api	zuständige Poolmaster für Datum
GET	/api/pools/mastered/{user}/{date}	auth:api	von Person betreute Pools

7.9. Berichte, Tabs und Hilfsdienste

Methode	Pfad	Schutz	Zweck
MATCH GET,POST	/api/report/{report}/{step}	auth:api	Reportschritte
GET	/api/tab/{tab}	auth:api	Tab-Inhalt
GET	/api/tab/{tab}/count	auth:api	Zähler für Tab
GET	/api/inbox	auth:api	Upload- oder Posteingangsübersicht
GET	/api/ministries/list	auth:api	Dienstekatalog
GET	/api/pixabay/query/{query}/{page?}	auth:api	Pixabay-Suche
GET	/api/rites/find/{query}	auth:api	Suche in Kasualien
DELETE	/api/occurence/{occurence}	auth:api	einzelnes Vorkommen löschen

7.10. Anmerkungen zur Pflege

- Öffentliche API-Routen dürfen nur noch mit ausdrücklicher fachlicher Begründung angelegt werden. Standard ist immer `auth:api`.
- Die Route-Definitionen allein beschreiben noch nicht alle Validierungsdetails. Für Payloads und Seiteneffekte sind die jeweiligen Controller und Requests maßgeblich.

8. OpenAPI und Schnittstellenpflege

Pfarrplaner enthält bereits eine technische Grundlage für eine maschinenlesbare API-Beschreibung.

8.1. Generator

Der OpenAPI-Build wird über diesen Artisan-Befehl erzeugt:

```
php artisan openapi:generate
```

Standardziel ist:

```
public/openapi.json
```

8.2. Technische Grundlage

- `app/Console/Commands/GenerateOpenApiSpec.php`
- `app/Http/Controllers/Api/OpenApiSpec.php`
- OpenAPI-Attribute direkt an API-Controllern

8.3. Veröffentlichung im Handbuch

Wenn `public/openapi.json` vorhanden ist, wird die Datei beim Build des technischen Handbuchs zusätzlich in die statische Handbook-Site kopiert. Damit kann sie gemeinsam mit der technischen Dokumentation ausgeliefert werden.

8.4. Empfohlener Pflegeprozess

1. API-Route anlegen oder ändern.
2. Controller und Request-Klassen anpassen.
3. OpenAPI-Attribute ergänzen oder aktualisieren.
4. `php artisan openapi:generate` ausführen.
5. Technisches Handbuch prüfen und gegebenenfalls Kapiteltexte aktualisieren.

8.5. Zielbild

Langfristig sollte jede öffentlich oder integrationsrelevant genutzte API-Funktion sowohl:

- in `openapi.json` beschrieben sein
- als erklärender Text im technischen Handbuch auftauchen

So bleibt die Dokumentation sowohl für Menschen als auch für Werkzeuge brauchbar.

8.6. Härtungsregel für neue Endpunkte

Bei neuen API-Routen gilt im Projekt jetzt die Regel:

- `/api` ist standardmäßig geschützt
- öffentliche Ausnahmen müssen ausdrücklich begründet und dokumentiert werden
- derzeit ist der Health-Check unter `/api/health` die vorgesehene öffentliche Ausnahme

Bei Änderungen an der API-Dokumentation ist deshalb immer mitzupflegen, ob ein Endpunkt geschützt oder bewusst öffentlich ist.

9. Stichwortverzeichnis

Dieses Stichwortverzeichnis bündelt technische Begriffe, damit Architektur, Betrieb und API schneller gefunden werden.

9.1. A

9.2. B

9.3. C

9.4. D

9.5. F

9.6. G

9.7. H

9.8. I

9.9. J

- JSON: API-Endpunkte

9.10. L

- Laravel: Überblick, Architektur und Domänen
- Laufzeit: Laufzeit, Build und Hintergrundprozesse
- Liturgie-API: API-Endpunkte

9.11. M

- Middleware: API-Authentifizierung
- MkDocs: Laufzeit, Build und Hintergrundprozesse

9.12. N

- Node.js: Laufzeit, Build und Hintergrundprozesse
- npm: Laufzeit, Build und Hintergrundprozesse

9.13. O

- Office-Ausgaben: Architektur und Domänen
- OpenAPI: API-Endpunkte, OpenAPI und Schnittstellenpflege

9.14. P

- PDF: Architektur und Domänen, Laufzeit, Build und Hintergrundprozesse
- Puppeteer: Überblick, Laufzeit, Build und Hintergrundprozesse

9.15. Q

- Queue: Laufzeit, Build und Hintergrundprozesse

9.16. R

- RoadRunner: Überblick, Laufzeit, Build und Hintergrundprozesse
- Routing: Architektur und Domänen, Dateien und Verzeichnisse

9.17. S

- Sanctum: API-Authentifizierung
- Scheduler: Laufzeit, Build und Hintergrundprozesse
- Scripts: Dateien und Verzeichnisse
- Storage: Dateien und Verzeichnisse

9.18. T

- TOC: Laufzeit, Build und Hintergrundprozesse

9.19. V

- Vite: Überblick, Laufzeit, Build und Hintergrundprozesse
- Vue: Überblick, Architektur und Domänen