Wasp & OpenSaaS Boilerplate - Technische Kenmerken

Complete uitleg van framework features, constraints en design decisions

Last Updated: 2025-11-25

📖Document Doel

Dit document legt uit **wat Wasp en OpenSaaS uniek maakt** en waarom bepaalde design decisions zijn gemaakt. Het is essentiële kennis voor developers die met deze stack werken.

Audience:

- Nieuwe developers (begrijpen waarom dingen anders werken)
- Experienced developers (van React/Node.js naar Wasp)
- Tech leads (architectuur beslissingen)

🎯Quick Overview

```
┌─────────────────────────────────────────────────────────────┐
│            TECH STACK LAYERS                │
└─────────────────────────────────────────────────────────────┘

Layer 4: Application (YOUR CODE)
┌──────────────────────────────────────────────────────────┐
│ Your Features (Tasks, Documents, Dashboard, etc.)    │
│ • React Components (.tsx)                    │
│ • Server Operations (.ts)                    │
│ • Prisma Schema (schema.prisma)              │
└──────────────────────────────────────────────────────────┘
            ↓
Layer 3: OpenSaaS Boilerplate
┌──────────────────────────────────────────────────────────┐
│ Pre-built Features                     │
│ • Authentication (email, social)         │
│ • Payment Integration (Stripe)           │
│ • Multi-language Support (i18next)        │
│ • Admin Dashboard (analytics)             │
│ • UI Components (ShadCN v2.3.0)           │
└──────────────────────────────────────────────────────────┘
            ↓
Layer 2: Wasp Framework (0.18.x)
┌──────────────────────────────────────────────────────────┐
│ Framework Features                     │
│ • Declarative Config (main.wasp)          │
│ • Auto-Generated API                   │
│ • Type-Safe Operations                 │
│ • Auto-Invalidation                    │
│ • Built-in Auth System                 │
└──────────────────────────────────────────────────────────┘
            ↓
Layer 1: Base Technologies
┌──────────────────────────────────────────────────────────┐
│ • React 18 (Frontend)                  │
│ • Node.js 20 (Backend)                 │
│ • Prisma 5.x (ORM)                     │
│ • PostgreSQL (Database)                │
│ • Vite (Build Tool)                    │
└──────────────────────────────────────────────────────────┘
```

🚀Wat is Wasp?

Definition

> **Wasp** is een full-stack web framework dat React (frontend) en Node.js (backend) **declartief configureert** via een DSL (Domain Specific Language).

In één zin: Wasp = React + Node.js + Prisma + **Auto-Generated Glue Code**

Hoe verschilt Wasp van normale React/Node apps?

Aspect	Traditional (React + Node.js)	Wasp Framework
API Design	Manual (Express routes, REST/GraphQL)	❌ **Auto-generated from config**
Type Sync	Manual (shared types, manual sync)	❌ **Auto-synced (server → client)**
Routing	Code-based (React Router setup)	✅ **Declarative (main.wasp)**
Auth	Manual (Passport, JWT, sessions)	✅ **Built-in (email, social, etc.)**
Database	Manual (Knex, TypeORM, raw queries)	✅ **Prisma (integrated)**
Operations	Manual (controllers, services)	✅ **TypeScript functions**
Client calls	Manual (fetch, axios, React Query)	✅ **Auto-generated hooks**
Dev setup	Multiple terminals (frontend, backend, DB)	✅ **One command (`wasp start`)**

Key insight: Wasp **eliminates boilerplate** by auto-generating API, types, and plumbing.

Core Wasp Concepts

1. Declarative Config (`main.wasp`)

Traditional approach:

```typescript
// server.js
app.post('/api/tasks', authMiddleware, async (req, res) => {
  // Validate, query DB, return JSON
})

// App.tsx
<Route path="/tasks" element={<TasksPage />} />
```

Wasp approach:

wasp

```wasp
// main.wasp - DECLARATIVE
route TasksRoute { path: "/app/tasks", to: TasksPage }
page TasksPage {
  authRequired: true,
  component: import { TasksPage } from "@src/pages/TasksPage"
}

action createTask {
  fn: import { createTask } from "@src/server/tasks/operations",
  entities: [Task]
}
```

Wasp auto-generates:

- ✅`/api/createTask` REST endpoint
- ✅Type-safe client hook: `createTask()`
- ✅React Router configuration
- ✅Auth middleware
- ✅Auto-invalidation when Task changes

Benefits:

- 🎯Single source of truth (main.wasp)
- 🎯No API design needed
- 🎯Types automatically flow
- 🎯Less code to maintain

---

2. Operations (Queries & Actions)

Operations = Server-side functions that become API endpoints

```typescript
// File: app/src/server/tasks/operations.ts
import type { CreateTask } from 'wasp/server/operations'
import { HttpError } from 'wasp/server'

export const createTask: CreateTask = async (args, context) => {
  // 1. Auth check
  if (!context.user) throw new HttpError(401)

  // 2. Validation
  if (!args.title) throw new HttpError(400, 'Title required')

  // 3. Database operation
  return await context.entities.Task.create({
    data: {
      title: args.title,
      userId: context.user.id
    }
  })
}
```

Wasp generates:

```typescript
// Client-side (auto-generated)
import { createTask } from 'wasp/client/operations'

// Usage in React component
await createTask({ title: 'New task' })
// → Calls /api/createTask
// → Returns type-safe Task object
// → Auto-invalidates queries
```

Key differences from traditional:

- ✅No Express routes
- ✅No manual JSON parsing
- ✅No manual type definitions
- ✅No React Query setup
- ✅No cache invalidation code

---

3. Entities (Database Models)

Entities = Prisma models configured in Wasp

prisma

```prisma
// File: app/schema.prisma
model Task {
  id          String   @id @default(uuid())
  title       String
  description String?
  status      String   @default("TODO")
  userId      String
  user        User     @relation(fields: [userId], references: [id])
  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt
}
```

In main.wasp:

wasp

```wasp
entity Task {=psl
  id          String   @id @default(uuid())
  title       String
  // ... (or import from schema.prisma)
psl=}
```

Wasp provides:

- ✅`context.entities.Task` in operations
- ✅Type-safe Prisma Client
- ✅Auto-migration commands
- ✅Type generation for TypeScript

Note: Since Wasp 0.14+, schema.prisma is recommended over inline entities.

---

4. Auto-Generated Types

Type flow: Server → Client (automatic)

```typescript
// Server operation (app/src/server/tasks/operations.ts)
export const getTasks: GetTasks = async (args, context) => {
  return await context.entities.Task.findMany()
  // Returns: Task[] (from Prisma)
}
```

Client code (auto-typed):

```typescript
// app/src/pages/TasksPage.tsx
import { useQuery } from 'wasp/client/operations'
import { getTasks } from 'wasp/client/operations'

export function TasksPage() {
  const { data: tasks } = useQuery(getTasks)
  //     ^^^^^^^^^^^^^ Automatically typed as Task[] | undefined

  return <div>{tasks?.map(task => ...)}</div>
}
```

Magic: Wasp generates TypeScript types from operations, no manual type files needed!

---

5. Built-in Authentication

Traditional auth setup:

```typescript
// 200+ lines of boilerplate
// - Passport.js setup
// - Session middleware
// - JWT signing/verification
// - Password hashing (bcrypt)
// - OAuth providers
// - Token refresh logic
```

Wasp auth setup:

```wasp
// main.wasp - ONE block
app myApp {
  auth: {
    userEntity: User,
    methods: {
      usernameAndPassword: {},
      google: { ... },
      github: { ... }
    },
    onAuthFailedRedirectTo: "/login"
  }
}
```

Wasp provides:

- ✅Password hashing (bcrypt)
- ✅Session management
- ✅Social OAuth (Google, GitHub, etc.)
- ✅Email verification
- ✅Password reset
- ✅`context.user` in operations
- ✅`useAuth()` hook in React

No manual implementation needed!

---

6. Auto-Invalidation

Traditional approach:

```typescript
// After mutation, manually invalidate cache
const mutation = useMutation(createTask, {
  onSuccess: () => {
    queryClient.invalidateQueries(['tasks'])
  }
})
```

Wasp approach:

typescript

```typescript
// Just call the action
await createTask({ title: 'New task' })
// Wasp automatically invalidates all queries that use Task entity!
```

How it works:

1. Action declares `entities: [Task]` in main.wasp
2. Wasp tracks which queries read Task
3. After action completes, Wasp auto-invalidates those queries
4. Components re-fetch automatically

Benefits: No manual cache management!

Wasp Architecture Diagram

```
┌─────────────────────────────────────────────────────────┐
│              WASP APPLICATION                │
└─────────────────────────────────────────────────────────┘

main.wasp (Configuration)
├─ app { ... }
├─ route TasksRoute { ... }
├─ page TasksPage { ... }
├─ query getTasks { ... }
└─ action createTask { ... }
      │
      ↓ (Wasp compiles)

.wasp/out/ (Generated Code)
├─ server/
│  ├─ src/
│  │  ├─ routes/ (Auto-generated REST API)
│  │  ├─ entities/ (Prisma Client wrapper)
│  │  └─ auth/ (Auth middleware)
│  └─ bundle/ (Node.js server)
│
└─ web-app/
   ├─ src/
   │  ├─ operations.ts (Type-safe client)
   │  ├─ auth.ts (useAuth hook)
   │  └─ router.tsx (React Router config)
   └─ bundle/ (React app)

Your Code (app/src/)
├─ server/
│  └─ tasks/
│     └─ operations.ts (Your logic)
├─ pages/
│  └─ TasksPage.tsx (Your UI)
└─ components/
   └─ TaskCard.tsx (Your components)
```

Key insight: Wasp generates the "plumbing" (API, types, routing), you write the "logic" (operations, UI).

🎨Wat is OpenSaaS?

Definition

> **OpenSaaS** is een **production-ready SaaS boilerplate** gebouwd op Wasp, met pre-built features zoals auth, payments, admin dashboard, en i18n.

In één zin: OpenSaaS = Wasp + Auth + Payments + Admin + UI Components + Best Practices

---

OpenSaaS Features

1. Authentication (Pre-built)

Included providers:

- ✅Email + Password (with verification)
- ✅Google OAuth
- ✅GitHub OAuth
- ✅Password reset flow
- ✅Email templates (SendGrid)

Usage:

```typescript
// Already configured - just use!
import { useAuth } from 'wasp/client/auth'

export function MyComponent() {
  const { data: user } = useAuth()

  if (!user) return <LoginPage />

  return <div>Welcome, {user.email}!</div>
}
```

Files:

```
app/src/auth/
├── LoginPage.tsx (Pre-built login UI)
├── SignupPage.tsx (Pre-built signup UI)
├── ForgotPasswordPage.tsx
├── VerifyEmailPage.tsx
└── components/ (Auth UI components)
```

---

2. Payment Integration (Stripe)

Pre-configured:

- ✅Stripe checkout
- ✅Subscription management
- ✅Webhook handling
- ✅Invoice generation
- ✅Payment methods

Usage:

```typescript
// Already integrated - just use!
import { createStripeCheckout } from 'wasp/client/operations'

const handleSubscribe = async () => {
  const { sessionUrl } = await createStripeCheckout({
    priceId: 'price_1234'
  })
  window.location.href = sessionUrl
}
```

Files:

```
app/src/payment/
├── PricingPage.tsx (Pre-built pricing page)
├── CheckoutPage.tsx
├── stripe/
│   ├── operations.ts (Stripe integration)
│   └── webhooks.ts (Webhook handlers)
└── components/ (Payment UI components)
```

---

3. Admin Dashboard

Pre-built analytics:

- ✅User metrics (signups, active users)
- ✅Revenue metrics (MRR, churn)
- ✅Charts (Chart.js integration)
- ✅Export data (CSV)

Usage:

```typescript
// Already built - just customize!
import { AdminDashboardPage } from '@src/admin/dashboards/AdminDashboardPage'

// Access at /admin/dashboard (admin role required)
```

Files:

```
app/src/admin/
├── dashboards/
│   ├── AnalyticsDashboardPage.tsx
│   └── UserDashboardPage.tsx
└── components/ (Dashboard components)
```

---

4. Multi-Language Support (i18n)

Pre-configured:

- ✅i18next integration
- ✅Language switcher component
- ✅Translation files (EN, NL, FR, etc.)
- ✅Language persistence (localStorage)

Usage:

```typescript
import { useTranslation } from 'react-i18next'

export function MyComponent() {
  const { t } = useTranslation()

  return (
    <div>
      <h1>{t('welcome.title')}</h1>
      <p>{t('welcome.description')}</p>
    </div>
  )
}
```

Files:

```
app/src/i18n/
├── i18n.ts (Configuration)
├── locales/
│   ├── en/
│   │   └── translation.json
│   ├── nl/
│   │   └── translation.json
│   └── fr/
│       └── translation.json
└── components/
    └── LanguageSwitcher.tsx
```

---

5. UI Components (ShadCN)

Pre-installed ShadCN v2.3.0:

- ✅40+ components (Button, Card, Dialog, etc.)
- ✅Tailwind CSS configured
- ✅Dark mode support
- ✅Accessible (ARIA)

Usage:

```typescript
import { Button } from '@/components/ui/button'
import { Card } from '@/components/ui/card'
import { Dialog } from '@/components/ui/dialog'

export function MyComponent() {
  return (
    <Card>
      <h1>My Card</h1>
      <Button>Click me</Button>
    </Card>
  )
}
```

Files:

```
app/src/components/ui/
├── button.tsx
├── card.tsx
├── dialog.tsx
├── dropdown-menu.tsx
├── input.tsx
└── ... (40+ components)
```

Note: ONLY ShadCN v2.3.0 (Tailwind v4 incompatible!)

---

6. Email Templates

Pre-built email templates:

- ✅Welcome email
- ✅Email verification
- ✅Password reset
- ✅Invoice email
- ✅Newsletter

Usage:

```typescript
import { sendWelcomeEmail } from '@src/email/operations'

await sendWelcomeEmail({
  to: user.email,
  name: user.name
})
```

Files:

```
app/src/email/
├── operations.ts (SendGrid integration)
└── templates/
    ├── welcome.tsx (React Email templates)
    ├── verification.tsx
    └── password-reset.tsx
```

OpenSaaS Directory Structure

```
app/
├── main.wasp (Wasp config)
├── schema.prisma (Database schema)
│
└── src/
    ├── auth/ (OpenSaaS: Pre-built auth)
    ├── payment/ (OpenSaaS: Stripe integration)
    ├── admin/ (OpenSaaS: Admin dashboard)
    ├── email/ (OpenSaaS: Email templates)
    ├── i18n/ (OpenSaaS: Multi-language)
    │
    ├── pages/ (YOUR product pages)
    ├── components/ (YOUR components)
    ├── server/ (YOUR operations)
    └── lib/ (YOUR utilities)
```

Principle: OpenSaaS provides **foundation**, you build **product features** on top.

🔧Belangrijke Wasp Constraints

1. Client/Server Separation (STRICT)

Rule: Client code CANNOT import server code

```typescript
// ❌WRONG - This will ERROR
// File: app/src/pages/TasksPage.tsx (CLIENT)
import { getTasks } from '../server/tasks/operations'
// ERROR: Module not found

// ✅CORRECT - Use Wasp operation
import { getTasks } from 'wasp/client/operations'
```

Why: Wasp enforces client/server boundary for security and bundle size.

---

2. Import Paths (Specific Rules)

Context	Rule	Example
main.wasp	Use `@src/...` alias	`import { Page } from "@src/pages/Page"`
.ts/.tsx files	Use **relative paths**	`import { Page } from "../pages/Page"`
UI components	Use `@/components/ui/...`	`import { Button } from "@/components/ui/button"`

Common mistakes:

```typescript
// ❌WRONG - @src in TypeScript files
import { TaskCard } from '@src/components/tasks/TaskCard'

// ✅CORRECT - Relative paths
import { TaskCard } from '../../components/tasks/TaskCard'

// ❌WRONG - wasp imports with @
import { Task } from '@wasp/entities'

// ✅CORRECT - wasp imports without @
import type { Task } from 'wasp/entities'
```

See: COMMON-PITFALLS.md#import-errors

---

3. Enum Runtime Values

Rule: Import enum TYPES from `wasp/entities`, RUNTIME VALUES from `@prisma/client`

```typescript
// ❌WRONG - Runtime value undefined!
import { UserRole } from 'wasp/entities'
if (user.role === UserRole.ADMIN) { /* undefined! */ }

// ✅CORRECT - Import runtime from Prisma
import type { User } from 'wasp/entities' // Type only
import { UserRole } from '@prisma/client' // Runtime values

if (user.role === UserRole.ADMIN) { /* ✅Works! */ }
```

Why: `wasp/entities` only exports TypeScript types, not JavaScript runtime values.

---

4. Database Migrations (Wasp Commands ONLY)

Rule: ALWAYS use `wasp db migrate-dev`, NEVER `prisma migrate` directly

```bash
# ❌WRONG - Bypasses Wasp type generation
npx prisma migrate dev --name "add field"

# ✅CORRECT - Wasp command
wasp db migrate-dev "add field"

# After migration, MANDATORY restart
./scripts/safe-start.sh
```

Why: Wasp needs to regenerate types after schema changes.

---

5. Operations Type Annotations (REQUIRED)

Rule: Always type-annotate operations with Wasp-generated types

```typescript
// ❌WRONG - No types
export const getTasks = async (args, context) => {
  return await context.entities.Task.findMany()
}

// ✅CORRECT - Type annotation
import type { GetTasks } from 'wasp/server/operations'

export const getTasks: GetTasks = async (args, context) => {
  return await context.entities.Task.findMany()
}
```

Why: Enables type-safe client hooks and catches errors at compile-time.

---

6. Email Access (Helper Required)

Rule: Use `getEmail(user)` helper, NOT `user.email`

```typescript
// ❌WRONG - Email not on User entity
const email = user.email // undefined or type error!

// ✅CORRECT - Use helper
import { getEmail } from 'wasp/auth/utils'
const email = getEmail(user)
```

Why: Wasp stores email in nested `auth.identities` structure, not directly on User.

7. Server Environment Variables ONLY

Rule: Server secrets MUST be in `.env.server`, client config in `.env.client`

```bash
# ❌WRONG - Server secret in client
# app/.env.client
OPENAI_API_KEY="sk-..." # Exposed to browser!

# ✅CORRECT - Server secrets in .env.server
# app/.env.server (NEVER commit!)
OPENAI_API_KEY="sk-..."
DATABASE_URL="postgresql://..."

# Client config in .env.client
# app/.env.client (safe to commit)
REACT_APP_PUBLIC_URL="http://localhost:3000"
```

Why: Client env vars are bundled in JavaScript (visible to users), server env vars stay on server.

🏗️Wasp Development Workflow

Traditional React/Node Workflow

```
┌──────────────────────────────────────────────────────────┐
│      TRADITIONAL WORKFLOW (Fragmented)           │
└──────────────────────────────────────────────────────────┘

Terminal 1: Frontend
$ npm run dev (React)

Terminal 2: Backend
$ npm run server (Node.js)

Terminal 3: Database
$ docker run postgres

Terminal 4: Type Generation
$ npm run generate:types (manual)

Terminal 5: API Documentation
$ npm run docs (Swagger, manual)

Total: 5 terminals, manual coordination
```

Wasp Workflow

```
┌──────────────────────────────────────────────────────────────┐
│       WASP WORKFLOW (Unified)                │
└──────────────────────────────────────────────────────────────┘

Terminal 1: Everything
$ wasp start

Auto-starts:
✅Frontend server (React + Vite)
✅Backend server (Node.js + Express)
✅Database (PostgreSQL Docker)
✅Type generation (automatic)
✅Hot reload (both frontend + backend)

Total: 1 command, zero coordination
```

Key benefits:

- 🎯One command starts everything
- 🎯Auto-restart on file changes
- 🎯Auto-type generation
- 🎯Zero manual coordination

🎨Wasp Design Philosophy

1. Full-Stack Single Developer

Traditional: Backend dev + Frontend dev (2 people)

Wasp: Full-stack dev (1 person owns complete feature)

Why possible:

- ✅Operations are simple TypeScript functions
- ✅Prisma query builder (not raw SQL)
- ✅Auto-generated API (no API design)
- ✅Auto-synced types (no manual contracts)
- ✅Built-in auth (no auth expertise needed)

Result: One developer can build UI + operations + database in same branch.

See: TEAM-STRUCTURE-AND-WASP-PHILOSOPHY.md

---

2. Feature-Based Structure (Vertical)

Traditional (Layer-based):

```
src/
├── frontend/ (ALL React code)
└── backend/ (ALL Node.js code)
```

Wasp (Feature-based):

```
src/
├── tasks/
│   ├── TasksPage.tsx (UI)
│   └── operations.ts (Server)
├── documents/
│   ├── DocumentsPage.tsx (UI)
│   └── operations.ts (Server)
```

Benefits:

- ✅Feature co-located (easy to find)
- ✅One developer owns complete feature
- ✅No cross-team coordination
- ✅Parallel development (no conflicts)

---

3. Declarative Over Imperative

Imperative (traditional):

```typescript
// Write HOW to do things
app.post('/api/tasks', async (req, res) => {
  // Auth check
  if (!req.user) return res.status(401).json(...)

  // Parse body
  const data = req.body

  // Validate
  if (!data.title) return res.status(400).json(...)

  // Query DB
  const task = await db.task.create(...)

  // Return JSON
  res.json(task)
})
```

Declarative (Wasp):

```wasp
// Declare WHAT you want
action createTask {
  fn: import { createTask } from "@src/server/tasks/operations",
  entities: [Task]
}
```

Benefits:

- ✅Less boilerplate
- ✅Wasp handles plumbing
- ✅Single source of truth
- ✅Less code to maintain

---

4. Convention Over Configuration

Wasp conventions:

- ✅`app/src/pages/` → Pages (auto-detected)
- ✅`app/src/server/` → Server code (auto-detected)
- ✅`app/schema.prisma` → Database schema (auto-detected)
- ✅`operations.ts` → Server operations (convention)
- ✅`main.wasp` → Configuration (single file)

Result: Minimal configuration, standard structure, easy onboarding.

🔄Data Flow (Complete Cycle)

```
┌─────────────────────────────────────────────────────────────┐
│           DATA FLOW: CLIENT → SERVER → DB                   │
└─────────────────────────────────────────────────────────────┘

1. USER ACTION (React Component)
   ┌─────────────────────────────────────┐
   │ TasksPage.tsx                       │
   │                                     │
   │ const handleCreate = async () => {  │
   │   await createTask({                │
   │     title: 'New task'               │
   │   })                                │
   │ }                                   │
   └─────────────────────────────────────┘
              │
              │ (1) Call operation
              ↓

2. WASP CLIENT (Auto-generated)
   ┌─────────────────────────────────────┐
   │ .wasp/out/web-app/src/operations.ts │
   │                                     │
   │ export const createTask = async (...) │
   │   // POST /api/createTask            │
   │   // Auto-invalidates queries       │
   │   return response.json()            │
   │ }                                   │
   └─────────────────────────────────────┘
              │
              │ (2) HTTP POST /api/createTask
              ↓

3. WASP SERVER (Auto-generated)
   ┌─────────────────────────────────────┐
   │ .wasp/out/server/src/routes/...     │
   │                                     │
   │ app.post('/api/createTask', ...)    │
   │   // Auth middleware (automatic)    │
   │   // Call your operation            │
   │ })                                  │
   └─────────────────────────────────────┘
              │
              │ (3) Call your operation
              ↓

4. YOUR OPERATION (Your code)
```

```
   ┌──────────────────────────────────────┐
   │ app/src/server/tasks/operations.ts   │
   │                                      │
   │ export const createTask = async (...) │
   │   if (!context.user) throw 401       │
   │   return context.entities.Task.create │
   │ }                                    │
   └──────────────────────────────────────┘
            │
            │ (4) Prisma query
            ↓

5. DATABASE (PostgreSQL)
   ┌──────────────────────────────────────┐
   │ INSERT INTO "Task" (id, title, ...)  │
   │ RETURNING *                          │
   └──────────────────────────────────────┘
            │
            │ (5) Return row
            ↓

6. RESPONSE FLOWS BACK
   Operation → Wasp Server → Wasp Client → React

7. AUTO-INVALIDATION (Wasp magic!)
   ┌──────────────────────────────────────┐
   │ Wasp detects: entities: [Task]       │
   │ → Invalidates all queries with Task  │
   │ → Components re-fetch automatically  │
   └──────────────────────────────────────┘
```

Key insight: Steps 2, 3, 6, 7 are **100% automatic** (Wasp-generated)!

📊 Tech Stack Details

Frontend Stack

Technology	Version	Purpose	Notes
React	18.x	UI Framework	Functional components only
TypeScript	5.x	Type Safety	Strict mode enabled
Vite	5.x	Build Tool	Fast HMR, dev server
Tailwind CSS	3.x	Styling	Utility-first CSS
ShadCN	v2.3.0	UI Components	Pre-built accessible components
React Router	6.x	Routing	Auto-configured by Wasp
React Query	4.x	Data Fetching	Wrapped by Wasp operations
i18next	23.x	i18n	Multi-language support

Backend Stack

Technology	Version	Purpose	Notes
Node.js	20.x	Runtime	LTS version
Express	4.x	Web Framework	Auto-configured by Wasp
Prisma	5.x	ORM	Type-safe database queries
PostgreSQL	16.x	Database	Docker container
Passport.js	0.7.x	Auth	Wrapped by Wasp auth
bcrypt	5.x	Password Hashing	Auto-handled by Wasp
SendGrid	8.x	Email	OpenSaaS integration

Development Stack

Technology	Version	Purpose	Notes
Wasp CLI	0.18.x	Framework	Core tool
Docker	Latest	Containers	Database, Redis
Git	Latest	Version Control	Multi-worktree setup
Vitest	1.x	Unit Testing	Fast, Vite-powered
Playwright	1.x	E2E Testing	Browser automation
ESLint	8.x	Linting	Code quality
Prettier	3.x	Formatting	Code style

🎯Wasp vs Alternatives

Framework	Type	Learning Curve	Full-Stack?	Auto-Generated API?	Built-in Auth?
Wasp	Full-Stack Framework	Low	✅Yes	✅Yes	✅Yes
Next.js	React Framework	Medium	⚠️Partial	❌No	❌No
Remix	React Framework	Medium	⚠️Partial	❌No	❌No
Blitz.js	Full-Stack Framework	Medium	✅Yes	✅Yes	⚠️Partial
RedwoodJS	Full-Stack Framework	High	✅Yes	✅Yes (GraphQL)	⚠️Partial
T3 Stack	Stack Template	High	✅Yes	❌No	⚠️Partial

Wasp strengths:

- ✅Lowest learning curve (declarative config)
- ✅Best type safety (auto-generated types)
- ✅Best DX (one command, hot reload)
- ✅Built-in auth (production-ready)

Wasp limitations:

- ⚠️Smaller ecosystem (fewer plugins)
- ⚠️Less flexible (opinionated structure)
- ⚠️Newer framework (less mature)

📚Quick Reference

Wasp Commands

```bash
# Development
wasp start                # Start all servers
wasp start db             # Start database only
wasp clean                # Clean generated files

# Database
wasp db migrate-dev "description"  # Create migration
wasp db studio            # Open Prisma Studio
wasp db seed              # Run seed functions

# Build & Deploy
wasp build                # Build for production
wasp deploy               # Deploy (with provider config)

# Testing
wasp test client          # Run client tests
wasp test server          # Run server tests
```

File Structure Quick Reference

```
project/
├── app/
│   ├── main.wasp              # Wasp config (routes, pages, operations)
│   ├── schema.prisma          # Database schema
│   ├── .env.server            # Server secrets (NEVER commit!)
│   ├── .env.client            # Client config (safe to commit)
│   │
│   └── src/
│       ├── pages/             # YOUR pages
│       ├── components/        # YOUR components
│       │   └── ui/            # ShadCN components (OpenSaaS)
│       ├── server/            # YOUR operations
│       ├── lib/               # YOUR utilities
│       │
│       ├── auth/              # OpenSaaS: Auth pages
│       ├── payment/           # OpenSaaS: Stripe
│       ├── admin/             # OpenSaaS: Admin dashboard
│       ├── email/             # OpenSaaS: Email templates
│       └── i18n/              # OpenSaaS: i18n
│
├── .wasp/                     # Generated code (auto, don't edit!)
├── scripts/                   # Helper scripts
└── tasks/                     # Task management
```

Import Patterns Quick Reference

```typescript
// Wasp imports
import type { Task } from 'wasp/entities'           // Types
import { getTasks } from 'wasp/client/operations'   // Client operations
import { useAuth } from 'wasp/client/auth'          // Auth hook
import { HttpError } from 'wasp/server'             // Server utilities

// Prisma imports (runtime values)
import { UserRole, TaskStatus } from '@prisma/client'

// UI components
import { Button } from '@/components/ui/button'

// Your code (relative paths)
import { TaskCard } from '../../components/tasks/TaskCard'
import { formatDate } from '../lib/utils'
```

🚨 Common Mistakes

Mistake	Why Wrong	Correct Approach
Import server code in client	Client/server separation	Use `wasp/client/operations`
Use `@src/` in .ts files	Only works in main.wasp	Use relative paths
Import enums from `wasp/entities`	Types only, no runtime	Use `@prisma/client`
Use `prisma migrate` directly	Bypasses Wasp type gen	Use `wasp db migrate-dev`
Forget restart after schema change	Types not regenerated	Always `./scripts/safe-start.sh`
Access `user.email` directly	Email not on User	Use `getEmail(user)` helper
Put secrets in `.env.client`	Exposed to browser	Use `.env.server`

See: COMMON-PITFALLS.md for complete list

🎓 Learning Resources

Official Documentation

- **Wasp Docs:** https://wasp.sh/docs/
- **OpenSaaS Docs:** https://docs.opensaas.sh/
- **Prisma Docs:** https://www.prisma.io/docs/

Project Documentation

- **DEVELOPMENT-WORKFLOW.md** - Complete workflow
- **CODE-ORGANIZATION.md** - File structure
- **TEAM-STRUCTURE-AND-WASP-PHILOSOPHY.md** - Philosophy
- **COMMON-PITFALLS.md** - Mistakes to avoid
- **SECURITY-RULES.md** - Security best practices

🎯 Summary

Wasp Key Features:

1. ✅Declarative config (main.wasp)
2. ✅Auto-generated API (no manual endpoints)
3. ✅Auto-synced types (server → client)
4. ✅Built-in auth (production-ready)
5. ✅Operations pattern (TypeScript functions)
6. ✅Full-stack single developer (one person owns features)
7. ✅One command setup (`wasp start`)

OpenSaaS Additions:

1. ✅Pre-built auth (email, social OAuth)
2. ✅Stripe integration (payments, subscriptions)
3. ✅Admin dashboard (analytics, metrics)
4. ✅Multi-language (i18n)
5. ✅UI components (ShadCN)
6. ✅Email templates (SendGrid)

Key Constraints:

1. ⚠️Client/server separation (strict)
2. ⚠️Import paths (main.wasp vs .ts files)
3. ⚠️Enum runtime values (from @prisma/client)
4. ⚠️Wasp commands only (no direct Prisma)
5. ⚠️Restart after schema changes (mandatory)

Result: Production-ready SaaS starter with minimal boilerplate!

Last Updated: 2025-11-25 **Version:** 1.0 **Maintained By:** Tech Lead Team