User Stories 2 Tasks

🧩 1. Ausgangspunkt: Die User Story

Eine gute **User Story** ist klein genug, um in einem Sprint umgesetzt zu werden, und beschreibt **was** der Benutzer will und **warum**.

Einfaches Beispiel (CRUD-Operation):

> **Als** Benutzer **möchte ich** meine gespeicherten Favoriten ansehen können, **um** schnell auf häufig genutzte Artikel zuzugreifen.

Komplexes Beispiel (mit Geschäftslogik):

> **Als** Fahrer einer Fahrgemeinschaft **möchte ich** eine angebotene Fahrt stornieren können, **damit** ich bei kurzfristigen Verhinderungen die Mitfahrer rechtzeitig informieren kann.
>
> **Geschäftsregeln:**
>
> - Stornierung nur möglich, wenn mindestens 2 Stunden vor Abfahrtszeit
> - Bei späterer Stornierung muss ein Begründungstext (min. 20 Zeichen) angegeben werden
> - Alle akzeptierten Mitfahrer werden automatisch per Push-Benachrichtigung informiert
> - Notfallkontakte der Mitfahrer werden per SMS benachrichtigt
> - Die Fahrt wird auf Status "storniert" gesetzt und ist nicht mehr sichtbar

---

🧱 2. Ziel der Task-Ableitung

Tasks sind die **technischen Arbeitsschritte**, die notwendig sind, um die Story vollständig umzusetzen. 👉 Sie beantworten also nicht mehr *„Was will der Nutzer?“*, sondern *„Wie setzen wir das um?“*.

🧭 3. Vorgehensweise (Best Practice)

Schritt 1: Story in Akzeptanzkriterien zerlegen

Nutze z. B. das **Gherkin-Format**:

Einfaches Beispiel (Favoriten):

> **Given** ein eingeloggter Benutzer, **When** er die Favoriten-Seite öffnet, **Then** werden alle gespeicherten Favoriten angezeigt.

Komplexes Beispiel (Fahrt stornieren):

> **Scenario 1: Rechtzeitige Stornierung (> 2h vor Abfahrt) Given** eine Fahrt mit Status "offen" und Abfahrtszeit in 3 Stunden **And** die Fahrt hat 2 akzeptierte Mitfahrer **When** der Fahrer die Fahrt storniert **Then** wird die Fahrt auf Status "storniert" gesetzt **And** alle Mitfahrer erhalten eine Push-

> Benachrichtigung **And** alle Notfallkontakte erhalten eine SMS **And** die Fahrt wird aus der Suche entfernt
>
> **Scenario 2: Kurzfristige Stornierung (< 2h vor Abfahrt) Given** eine Fahrt mit Abfahrtszeit in 1 Stunde **When** der Fahrer ohne Begründung stornieren möchte **Then** wird ein Fehler angezeigt: "Begründung erforderlich" **When** der Fahrer mit Begründung (min. 20 Zeichen) storniert **Then** wird die Stornierung durchgeführt **And** die Begründung wird in der Benachrichtigung mitgesendet
>
> **Scenario 3: Bereits gestartete Fahrt Given** eine Fahrt mit Abfahrtszeit in der Vergangenheit **When** der Fahrer die Fahrt stornieren möchte **Then** wird ein Fehler angezeigt: "Fahrt kann nicht mehr storniert werden"

Diese Kriterien helfen, die Story **in prüfbare Teilaspekte** zu gliedern.

Schritt 2: Fachliche und technische Analyse

Gemeinsam im Team überlegen:

- Welche **Frontend-Komponenten** werden benötigt?
- Welche **Backend-Endpunkte** oder **Datenbankänderungen**?
- Welche **Geschäftslogik** muss im Middle-Tier (Application Server) implementiert werden?
- Welche **Domain Services** oder **Use Cases** sind notwendig?
- Welche **externen Services** müssen integriert werden (Push-Notifications, SMS)?
- Welche **Tests** müssen geschrieben werden?
- Welche **Dokumentation** oder **Review-Schritte** sind erforderlich?

Schritt 3: Tasks formulieren

Jeder Task beschreibt **eine abgeschlossene Arbeitseinheit**, idealerweise in 0.5–1 Tag umsetzbar.

Beispiel-Tasks zu einfacher Story (Favoriten):

Bereich	Beispiel-Task	Beschreibung
Frontend	UI-Komponente "Favoritenliste" erstellen	Anzeige der gespeicherten Favoriten in einer Liste
Backend	REST-Endpoint `/favorites` implementieren	Gibt alle Favoriten des Users zurück
Datenbank	Tabelle `favorites` anlegen	Speichert Zuordnung User ↔ Artikel
Testing	Unit Tests für Favoriten-Controller schreiben	Prüft CRUD-Operationen
Review	UI-Design-Review durchführen	Visuelle Konsistenz prüfen

Beispiel-Tasks zu komplexer Story (Fahrt stornieren) - mit Geschäftslogik:

Bereich	Task	Beschreibung	Geschäftslogik
Domain Model	Aggregat "Fahrt" erweitern	Methode `cancelRide(reason?, currentTime)` hinzufügen	✅ Invariante: Status muss "offen" sein
Domain Model	Value Object "CancellationReason" erstellen	Validierung: min. 20 Zeichen, max. 500 Zeichen	✅ Business Rule: Required wenn < 2h
Application Layer	Use Case "CancelRideUseCase" implementieren	Orchestriert Geschäftslogik und externe Services	✅ **Zentrale Geschäftslogik**
Business Logic	Zeitfenster-Validierung implementieren	Prüft: `departureTime - currentTime >= 2h`	✅ Core Business Rule
Business Logic	Status-Transition-Logik implementieren	Erlaubte Übergänge: "offen" → "storniert"	✅ State Machine
Business Logic	Benachrichtigungs-Orchestrierung	Entscheidet: Push + SMS basierend auf Zeitfenster	✅ Workflow-Logic
Domain Service	NotificationService erweitern	Methode `notifyRideCancellation(ride, reason?)`	✅ Multi-Step Process
Infrastructure	Push-Notification-Adapter implementieren	Integration mit FCM/APNs	⚙️ Technical
Infrastructure	SMS-Adapter implementieren	Integration mit SMS-Provider	⚙️ Technical
Backend	REST-Endpoint `DELETE /rides/{id}/cancel`	Body: `{ reason?: string }`	🔌 API Layer
Backend	DTO "CancelRideRequest" erstellen	Validierung der Eingabedaten	🔌 API Layer
Frontend	Stornierungsdialog mit Begründungsfeld	Conditional Rendering basierend auf Zeitfenster	🎨 UI
Frontend	Fehlerbehandlung für Stornierung	Anzeige von Validierungsfehlern	🎨 UI
Database	Migration: `cancelled_at` Feld hinzufügen	Timestamp der Stornierung	💾 Data
Database	Migration: `cancellation_reason` Feld	Text, nullable	💾 Data

Bereich	Task	Beschreibung	Geschäftslogik
Testing	Unit Tests für `Fahrt.cancelRide()`	Prüft Geschäftsregeln (Zeitfenster, Reason)	✅ Domain Logic
Testing	Integration Tests für CancelRideUseCase	Prüft Orchestrierung inkl. Notifications	✅ Application Logic
Testing	E2E Tests für Stornierungsflow	Prüft kompletten User Journey	🧪 Full Stack
Documentation	API-Dokumentation aktualisieren	Swagger: DELETE /rides/{id}/cancel	📚 Docs

Legende:

- ✅ **Geschäftslogik** (Business Rules, Domain Logic)
- ⚙️ **Technische Infrastruktur** (External Services)
- 🔌 **API Layer** (REST, DTO, Validation)
- 🎨 **Presentation Layer** (UI, UX)
- 💾 **Data Layer** (Database, Persistence)
- 🧪 **Testing** (Unit, Integration, E2E)
- 📚 **Documentation**

Erklärung: Wo liegt die Geschäftslogik?

Die **Geschäftslogik** (Business Logic) wird im **Middle-Tier** (Application Server) umgesetzt:

1. **Domain Model (Domain Layer)**
   - `Fahrt.cancelRide()` - Prüft Invarianten (Status, Zeitfenster)
   - `CancellationReason` - Validiert Begründungstext
2. **Application Layer (Use Case Layer)**
   - `CancelRideUseCase` - **Orchestriert** den gesamten Prozess:

```
class CancelRideUseCase {
  async execute(rideId: string, reason?: string, userId: string) {
    // 1. Fahrt laden
    const ride = await this.rideRepository.findById(rideId);

    // 2. Autorisierung prüfen
    if (ride.driverId !== userId) throw new ForbiddenError();

    // 3. Geschäftslogik: Stornierung durchführen
    const currentTime = new Date();
    ride.cancelRide(reason, currentTime); // ✅ Domain Model
entscheidet

    // 4. Persistieren
```

```
    await this.rideRepository.save(ride);

    // 5. Domain Event: RideCancelled publishen
    await this.eventBus.publish(new RideCancelledEvent(ride, reason));

    // 6. Benachrichtigungen orchestrieren
    await this.notificationService.notifyRideCancellation(ride,
reason);
  }
}
```

3. **Domain Service (Domain Layer)**
   - `NotificationService` - Entscheidet, welche Benachrichtigungen versendet werden

```
class NotificationService {
  async notifyRideCancellation(ride: Ride, reason?: string) {
    const passengers = await this.getAcceptedPassengers(ride);

    // Push-Benachrichtigungen an Mitfahrer
    for (const passenger of passengers) {
      await this.pushAdapter.send(passenger.userId, {
        title: "Fahrt storniert",
        body: reason ? `Grund: ${reason}` : "Leider wurde die Fahrt
storniert.",
        deepLink: `/rides/${ride.id}`
      });

      // SMS an Notfallkontakte
      const emergencyContacts = await
this.getEmergencyContacts(passenger);
      for (const contact of emergencyContacts) {
        await this.smsAdapter.send(contact.phoneNumber,
          `Die Fahrt von ${ride.driver.name} um ${ride.departureTime}
wurde storniert.`
        );
      }
    }
  }
}
```

Wichtig: Die **Geschäftsregeln** (z.B. Zeitfenster-Prüfung, Status-Validierung) gehören ins **Domain Model**, nicht in Controller oder UI!

Schritt 4: Definition of Done prüfen

Überprüfe, ob alle Tasks zusammen die **Akzeptanzkriterien** erfüllen und die **Definition of Done (DoD)** des Teams abdecken — z. B. Codequalität, Tests, Dokumentation, Review etc.

🧠 4. Best Practices zusammengefasst

✅ **Teamarbeit**: Tasks werden *im Sprint Planning oder Refinement Meeting* gemeinsam erstellt. ✅ **Kleine Tasks**: Lieber mehr kleine Tasks als wenige große – bessere Nachvollziehbarkeit. ✅ **Klar formuliert**: Ein Task sollte ein **konkretes Ergebnis** haben („UI-Komponente erstellt" statt „Frontend gemacht"). ✅ **Nachvollziehbar verlinkt**: Jeder Task ist einer User Story eindeutig zugeordnet. ✅ **Automatisierte Tests** immer als Teil der Tasks betrachten. ✅ **Geschäftslogik im Domain Model**: Business Rules gehören ins Domain Model, nicht in Controller oder UI. ✅ **Separation of Concerns**:

- **Domain Layer** = Was (Geschäftsregeln)
- **Application Layer** = Wie (Orchestrierung)
- **Infrastructure Layer** = Womit (Technische Details) ✅ **Use Cases für komplexe Logik**: Bei komplexen Stories Use Cases für Orchestrierung verwenden. ✅ **Domain Events**: Für asynchrone Prozesse (z.B. Benachrichtigungen) Domain Events nutzen.

🧰 Beispiel-Template für Task-Ableitung

Du kannst dieses Schema im Unterricht oder Teammeeting nutzen:

Einfache Story (CRUD):

Story	Akzeptanzkriterium	Mögliche Tasks
Als Nutzer möchte ich ...	When ... Then ...	1. Frontend-Komponente 2. API-Endpunkt 3. Datenbankänderung 4. Tests 5. Review

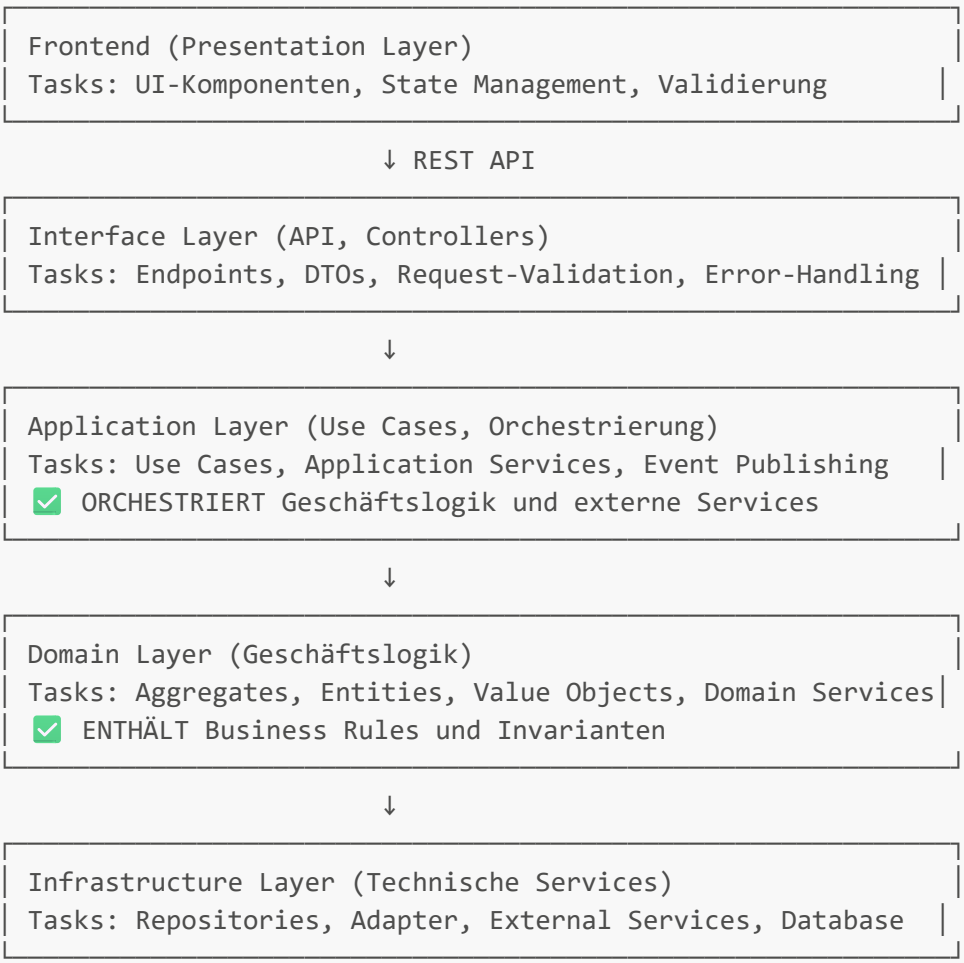
Komplexe Story (mit Geschäftslogik):

Story	Akzeptanzkriterium	Layer	Mögliche Tasks
Als X möchte ich Y, damit Z	**Scenario 1:** When ... Then ...	**Domain Layer**	1. Aggregat-Methode mit Business Rules 2. Value Objects für Validierung 3. Domain Services
	Scenario 2: When ... Then ...	**Application Layer**	4. Use Case für Orchestrierung 5. DTO für Request/Response 6. Event Publishing
	Scenario 3: When ... Then ...	**Infrastructure**	7. Adapter für externe Services 8. Repository-Implementierung
		Interface Layer	9. REST-Endpoint 10. Request-Validation
		Frontend	11. UI-Komponente 12. State Management

Story	Akzeptanzkriterium	Layer	Mögliche Tasks
		Testing	13. Unit Tests (Domain) 14. Integration Tests (Use Case) 15. E2E Tests

📐 Architektur-Schichten und ihre Tasks

Bei komplexen Stories mit Geschäftslogik sollten Tasks entlang der **Architektur-Schichten** strukturiert werden:



Wichtig: Geschäftslogik gehört **immer** in den **Domain Layer** oder **Application Layer**, niemals in Controller oder UI!