Inhalt

[2 Technologiestack 3](#_Toc393391102)

[2.1 Allgemein 3](#_Toc393391103)

[2.2 Programmiersprache 3](#_Toc393391104)

[2.3 Application-Server 3](#_Toc393391105)

[2.4 Datenbank 3](#_Toc393391106)

[3 Tooling (Development Infrastructure) 4](#_Toc393391107)

[3.1 Entwicklungsumgebung 4](#_Toc393391108)

[3.2 Code-Repository 4](#_Toc393391109)

[3.3 Build- und Dependency-Management 4](#_Toc393391110)

[3.4 Artifact Repository 4](#_Toc393391111)

[3.5 Build-Server 4](#_Toc393391112)

[3.6 Testframework 4](#_Toc393391113)

[4 Projektmanagement 5](#_Toc393391114)

[5 Releasemanagement 5](#_Toc393391115)

[5.1 Branching 5](#_Toc393391116)

[5.2 Bezeichnung der Branches 5](#_Toc393391117)

[5.3 Branching durch Build-Server beim Release 5](#_Toc393391118)

[5.4 Bezeichnung der Releases / Snapshots 6](#_Toc393391119)

[5.5 Releases im SCM 6](#_Toc393391120)

[6 Coding-Conventions 7](#_Toc393391121)

[6.1 Benamung von Klassen, Methoden und Variablen 7](#_Toc393391122)

[6.1.1 Sprache 7](#_Toc393391123)

[6.1.2 Kommentare 7](#_Toc393391124)

[6.1.3 Benamung von Services, Beans und Validatoren 7](#_Toc393391125)

[6.1.4 Benamung von Servicemethoden 7](#_Toc393391126)

[6.2 Benamung von Datenbanktabellen 7](#_Toc393391127)

[6.3 Codeformatierung 8](#_Toc393391128)

[6.4 REST-Services 8](#_Toc393391129)

[6.5 Kommunikation zwischen Client und Server via REST 9](#_Toc393391130)

[6.6 Definition und Verwendung von NamedQueries 9](#_Toc393391131)

[7 Projektstruktur 10](#_Toc393391132)

[7.1 Aufbau des Projekts 10](#_Toc393391133)

[7.2 Abhängigkeiten der Komponenten untereinander 10](#_Toc393391134)

[7.3 Maven-Module 11](#_Toc393391135)

[8 Directory und package layout 11](#_Toc393391136)

[8.1 Directory layout 11](#_Toc393391137)

[8.2 Package layout 11](#_Toc393391138)

[9 Validierung und Fehlerbehandlung 12](#_Toc393391139)

[10 Testen 14](#_Toc393391140)

# Technologiestack

Wie ist die Architektur aufgebaut und welche konkreten Technologien / Frameworks werden eingesetzt?

## Allgemein

* Client-Server-Architektur
* Client nur rudimentär entwickelt mit JavaFX 8
* Kommunikation via REST

## Programmiersprache

Java 8

## Application-Server

Wildfly (JBoss 8) mit JEE7

## Datenbank

MySQL 5.6

Datenbankanbindung via JDBC (JPA)

# Tooling (Development Infrastructure)

## Entwicklungsumgebung

Keine Festlegung, präferiert: Eclipse Kepler (Alternativen: IntelliJ, NetBeans, RAD)

## Code-Repository

Link zum zentralen Git-Repository: [git@github.com:mfroehlich/flightplan2.git](mailto:git@github.com:mfroehlich/flightplan2.git)

Infos zu Git: CheatSheet: <http://ndpsoftware.com/git-cheatsheet.html>

## Build- und Dependency-Management

Maven 3

## Artifact Repository

Nexus

Info: in einem konkreten Projekt müsste hier festgelegt werden, wie man die Nexus-Pfade im POM-file hinterlegt.

## Build-Server

Jenkins

Info: in einem konkreten Projekt müsste hier der Link zum Jenkins hinterlegt und festgelegt werden, wie Jenkins-Jobs zu definieren sind.

## Testframework

Arquillian (JUnit) + Shrinkwrap

# Projektmanagement

**TODO** Organisation der zu implementierenden Arbeitspakete mittels JIRA festlegen

* Vorgehensmodell festlegen
* Pro Release-Unit ein JIRA-Projekt

# Releasemanagement

## Branching

* Hauptentwicklung findet auf dem Trunk statt.
* Entwicklung von Features auf Branch (ausgehend vom Stand, auf dem das Feature entwickelt werden soll)  
  🡺 WICHTIG: sobald möglich, sollen Features in die Hauptentwicklungslinie zurückgemerged werden.
* Parallele Entwicklung mehrerer Features auf mehreren Branches  
  🡺 je nach zeitlicher Abfolge des Livegangs der Features kann erst bei Bedarf zurückgemerged werden
* Stabilisierung und Livegang eines Stands soll auf dediziertem Branch erfolgen.  
  🡺 dort erfolgen auch nachfolgende Bugfixes
* Grundsätzlich: nicht mehr verwendete Branches sollen gelöscht werden!

## Bezeichnung der Branches

**Release-/Bugfix-Branch**: RELEASE\_%Majorversion%.%Minorversion)

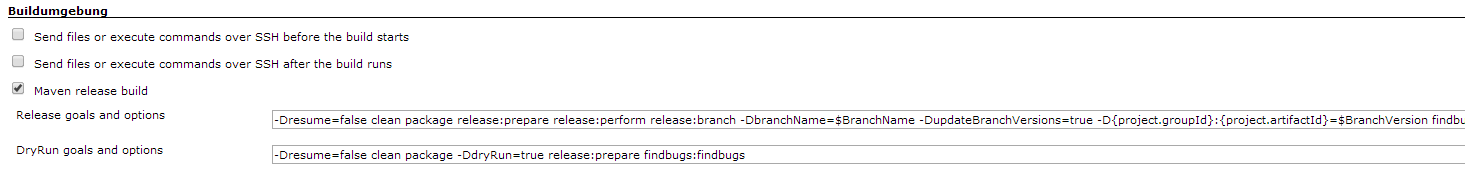
🡺z.B. RELEASE\_01.04

**Feature-Branch**: FEATURE\_%JIRA-ID%\_%Majorversion%.%Minorversion%  
wobei JIRA-ID die Nummer des JIRA-Tickets ist, mit dem das Feature verwaltet wird

🡺z.B. FEATURE\_FP-003\_01.04

## Branching durch Build-Server beim Release

Jenkins so konfigurieren, dass er beim Release automatisch - wenn benötigt - Branches im SCM zieht (und dort das Release baut).



Branching mit Maven im Jenkins

## Bezeichnung der Releases / Snapshots

* 2-stellige Zahl für das Major-Release
* 2-stellige Zahl für das Minor-Release
* 2-stellige Zahl für die Fixversion
* Optional angehängt: -SNAPSHOT

Beispiele: 01.00.00-SNAPSHOT, 01.01.02

## Releases im SCM

* Jede Release-Unit wird als eigenes Repository im Git abgebildet
* Bezeichnung der Repositories soll identisch sein mit dem Namen der Release-Unit

# Coding-Conventions

## Benamung von Klassen, Methoden und Variablen

### Sprache

Klassen und Variablennamen sind grundsätzlich in englischer Sprache zu vergeben. Domänenspezifische Begriffe oder Eigennamen können in deutscher Sprache bleiben, um Verwirrungen zu vermeiden.

### Kommentare

Kommentierung muss in englischer Sprache erfolgen. Kommentiert werden müssen alle public-Methoden und alle Klassen. Im Javadoc soll immer der Autor für Rückfragen angegeben sein.

### Benamung von Services, Beans und Validatoren

Service-Interfaces haben immer das Suffix „Service“, die implementierenden EJBs heißen exakt genauso, enden aber auf „Bean“, z.B. das Service-Interface PilotService wird implementiert durch die EJB PilotBean.

Ein Validator für ein Service-Interface soll heißen wie das Interface mit angehängtem Suffix „ValidationDecorator“. Der Validator des PilotService muss damit PilotServiceValidationDecorator heißen.

### Benamung von Servicemethoden

Die 4 CRUD-Methoden sollen folgendermaßen benannt werden: (Beispiel für Pilot)

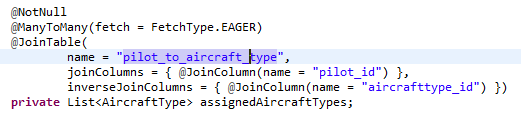
* Pilot erzeugen: createPilot(Pilot pilot)
* Einen speziellen Pilot laden: loadPilotById(String pilotId)
* Alle Piloten laden: loadAllPilots()
* Pilot aktualisieren: updatePilot(Pilot pilot)
* Pilot löschen: deletePilotById(String pilotId)

Somit sind die 4 einheitlich zu verwendenden Verben: create, load, update, delete.

## Benamung von Datenbanktabellen

Der Bezeichner einer Datenbanktabelle ist immer gleich dem Namen der Entität, welche sie enthält. Die CamelCase-Schreibweise einer Java-Klasse wird dabei allerdings aufgelöst und die Teilworte durch Unterstriche getrennt, z.B. AircraftType -> aircraft\_type.

JoinTables enthalten die Namen der gejointen Entitäten, verbunden durch das Bindewort „to“, z.B. pilot\_to\_aircraft\_type.



## Codeformatierung

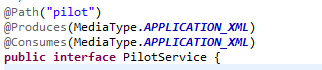
Für die einheitliche Formatierung sollen die Formatter.xml und cleanup.xml in die Entwicklungsumgebung importiert werden.

Anmerkung: im richtigen Projekt würde ich diese irgendwo bereitstellen.

## REST-Services

Die Service-Interfaces können direkt annotiert werden, um als REST-Service zur Verfügung gestellt zu werden.

Für die Einheitlichkeit ist als @Path-Name an einem REST-Service immer der (kleingeschriebene!) Name des Service-Interfaces (ohne „Service“-Suffix) zu setzen, d.h. der PilotService setzt @Path(„pilot“).



@Produces und @Consumes sind direkt auf Klassenebene zu setzen.



Die Authorisierungsannotation @RolesAllowed wird auch direkt auf den ServiceMethoden gesetzt, da RestEasy für die Authorisierung verwendet wird.

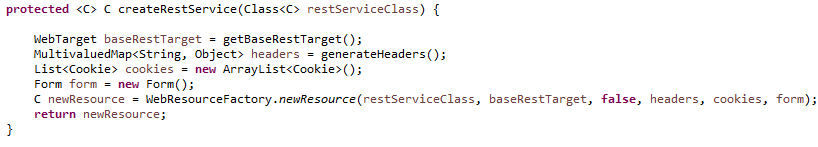


Die Übergabe von Path-Parametern erfolgt durch anfügen des Variablennamens an den Path + „/“ + Template {Variablenname}, d.h. @Path wird immer folgendermaßen ergänzt: @Path{„%VAR\_NAME%/{%VAR\_NAME%}), und der Parameter in der Methode wird folgendermaßen annotiert: @PathParam(„%VAR\_NAME%“).

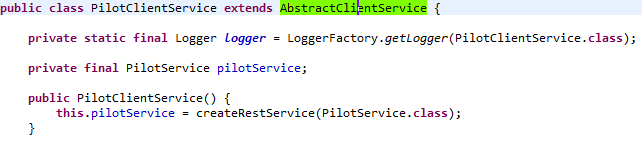
## Kommunikation zwischen Client und Server via REST

Client und Server kommunizieren über REST mit Hilfe eines Dynamic Proxies.

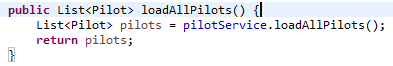
Der Client kann Serveraufrufe ausführen, indem er die durch den Server bereitgestellten Server-Interfaces über einen Dynamic Proxy anspricht. Hierfür existiert im Client die Klasse AbstractClientService, welche die Methode createRestService(…) bereitstellt:



AbstractClientService muss clientseitig immer dann von einer Klasse erweitert werden, wenn diese via Dynamic Proxy ein Server-Interface ansprechen möchte. So erweitert z.B. der PilotClientService die Klasse AbstractClientService und stellt eine Verbindung zum Server-Interface PilotService bereit:

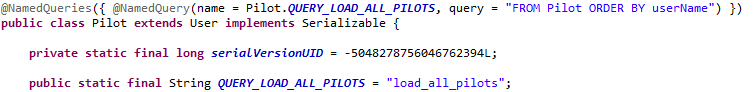


Die Methoden in PilotClientService können die Methoden des PilotService dann direkt aufrufen:



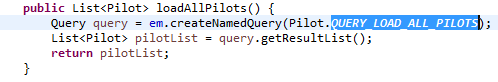
## Definition und Verwendung von NamedQueries

Wenn möglich, sollen NamedQueries verwendet werden, um JPQL-Anweisungen zu definieren.



Dabei soll der „name“ der Named-Query als Konstante innerhalb der Entität definiert werden. (siehe Screenshot: Pilot.QUERY\_LOAD\_ALL\_PILOTS.

Diese Konstante kann dann bei der Erzeugung der NamedQuery in der EJB ebenfalls herangezogen werden:



# Projektstruktur

## Aufbau des Projekts

Das Projekt ist ein Multi-Module-Project. Das Hauptprojekt ist ein POM-Projekt namens „flightplan“. Dieses legt in seinem POM-file die Versionen der dependencies (dependencyManagement) für alle Kinderprojekte fest.

Es gibt mehrere Domänen (Pilot, Plane, Reservation, User), welche jeweils zu eigenen Komponenten zusammengefasst sind. Eine Komponente beschreibt ein öffentliches Interface (API) und die zugehörigen Implementierungen (SPI).

In der Projektstruktur ist eine Komponente für eine Domäne ebenfalls als Maven-Modul realisiert, welches selbst 2 Kinder-Module hält. Eine Komponente ist somit als Child-Modul dem Hauptprojekt flightplan untergeordnet und besitzt selbst 2 child-Module, z.B. die Domäne „pilot“ ist unterhalb von flightplan als Maven-POM-Projekt „pilot“ aufgehängt und besitzt selbst 2 Module:

* pilot-client: API (öffentliche Schnittstelle)
* pilot-service: SPI (Implementierung der Schnittstelle)

Es gelten folgende Regeln:

* Jede Domäne findet sich in einer eigenen Komponente wieder
* Im Client-Modul der Komponente ist das Datenmodell und das öffentliche Service-Interface (Exceptions etc)
* Im Service-Modul befinden sich die zugehörigen Implementierungen.

## Abhängigkeiten der Komponenten untereinander

**WICHTIG:** Die Komponenten dürfen sich untereinander nur Abhängigkeiten auf ihre APIs haben, d.h. die Implementierung eines Service-Interfaces muss über CDI aufgelöst werden.

Dass dies funktioniert, muss das EAR-Projekt unterhalb von flightplan alle SPI-Module der Komponenten als Abhängigkeit definieren. Das EAR-Projekt ist aber das einzige Projekt, welches die SPI-Module als Abhängigkeit definieren darf.

## Maven-Module

Jedes Maven-Projekt muss eine GroupId und eine ArtifactId definieren.

Die ArtifactId eines Projekts soll immer identisch sein mit dem Namen des Projekts selbst.

Die GroupId soll die Hierarchie in der Projektstruktur wiederspiegeln. So definiert ein Unterprojekt als GroupId die GroupId des Parentprojekts + die ArtifactId des Parentprojekts.

Das Hauptprojekt flightplan definiert

* als GroupId: com.prodyna.pac
* als ArtifactId: flightplan (=Projektname)

Für das Unterprojekt pilot ergibt sich damit

* als GroupId: com.prodyna.pac.flightplan (GroupId+ArtifactId des Parent-Projekts)
* als ArtifactId: pilot (=Projektname)

Für das Unterprojekt pilot-client von pilot ergibt sich damit

* als GroupId: com.prodyna.pac.flightplan.pilot
* als ArtifactId: pilot-client

# Directory und package layout

## Directory layout

Maven gibt die Struktur vor:

* Src/main/java
* scr/main/resources
* src/test/java
* src/test/resources
* target
* pom.xml

## Package layout

Beispiel:   
flightplan/src/main/java/com/prodyna/pac/flightplan/pilot/entity/

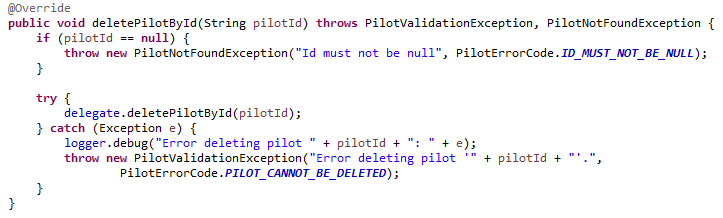
* Project root path: flightplan /
* Maven Directory layout: src/main/java/
* Root application package: com/prodyna/pac/flightplan/  
  (reverse domain name + app-name)
* Inner application packages:
  + pilot/entity/
  + pilot/exception
  + pilot/service
  + pilot/service/validation

**INFO**: Sonar kann zusätzlich die Einhaltung sauberer Abhängigkeiten zwischen den Paketen prüfen.

# Validierung und Fehlerbehandlung

Die Validierung soll in @Decorator-Klassen stattfinden, welche die jeweilige EJB dekoriert. Wie in 6.1.3 beschrieben, wird die PilotBean durch den PilotServiceValidationDecorator dekoriert und alle Methoden der PilotBean werden somit vorher deren Aufruf durch den Decorator validiert.

Im Folgenden die Methode deletePilotBId(String pilotId) im ValidationDecorator des PilotService:



Die Methode führt zunächst ein paar Validierungen durch und delegiert den Aufruf erst danach an den eigentlichen Service. So ist sichergestellt, dass die Validierung vor dem Aufruf des Service stattfindet.

Der Serviceaufruf selbst ist ebenfalls in diesem Fall in einem try-Block, so dass im Fehlerfall auch darauf mit einer entsprechenden Exception reagiert werden kann.

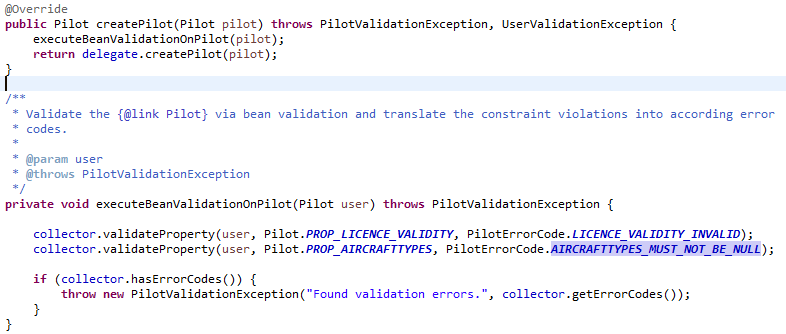
**Es gibt 2 Arten von Exceptions:**

* Fachliche Exceptions: FunctionalException (checked)
* Technische Exceptions: TechnicalException (runtime)

Technische Exceptions sind RuntimeExceptions, da diese nur geworfen werden, wenn ein technischer Fehler vorliegt, welcher durch den Benutzer nicht behoben werden kann. Im Gegensatz dazu sind fachliche Exceptions checked Exceptions, da diese explizit nach der Validierung von z.B. Benutzereingaben geworfen werden. Somit können FunctionalExceptions durch den Benutzer provoziert werden und es wird eine explizite Fehlerbehandlung durch das System eingeleitet, weshalb diese Exceptions sich auch in den Methodensignaturen wiederspiegeln sollte.

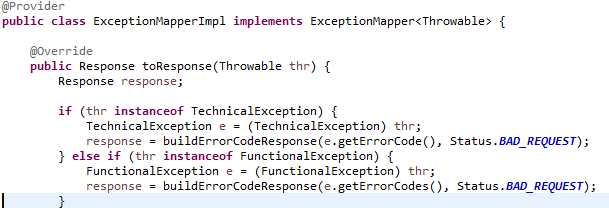
FunctionalExceptions und TechnicalExceptions können einen oder mehrere ErrorCode-Objekte halten. Ein ErrorCode steht für einen spezifischen Fehler und identifiziert sich als String. Jede Domäne (pilot, plane, …) erweitert die Klasse ErrorCode und definiert dort ihre eigenen domänenspezifischen ErrorCodes, die in der jeweiligen Domäne auftreten können.

Tritt ein Fehler auf (z.B. beim Validieren in einem ValidationDecorator), so wird z.B. eine FunctionalException geworfen und dieser eine Liste von ErrorCodes übergeben.



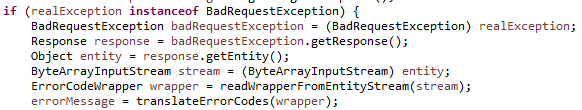
PilotValidationException ist eine FunctionalException, und diese enthält im obigen Beispiel alle ErrorCodes, die der Collector eingesammelt hat.

Geworfene Exceptions werden „auf dem Weg zum Client“ von der Klasse ExceptionMapperImpl gefangen und in REST-Responses transformiert:



Die in der Exception enthaltenen ErrorCodes werden dabei als Entity in das Response-Objekt gesetzt und der http-Statuscode 403 gesetzt.

Auf der Clientseite wird dann eine BadRequestException geworfen, welche die ErrorCodes ebenfalls beinhaltet. Eine zentrale Stelle, die Klasse ExceptionHandler, fängt die Exception, liest die ErrorCodes aus und übersetzt diese in lesbare Fehlermeldungen:



# Testen

* Vorbedingungen, bevor Code in das zentrale Repository eingecheckt werden darf:
  + Die entwickelten Features / Bugfixes müssen durch lokale Entwicklertests geprüft worden sein
  + Alle JUnit-Tests müssen erfolgreich laufen.
* Testabdeckung:
  + Für jede public-Methode einer EJB muss mindestens ein Arquillian-Test existieren, so dass die nach außen bereitgestellte Funktionalität sichergestellt und konserviert wird.
  + Für jede über REST verfügbare Methode muss ein REST-Test existieren, welcher mit @RunAsClient die Funktionalität „von außen“ testet.
* Zusätzlich: Einrichten eines Build-Servers für automatisiertes Ausführen der JUnit-Tests nach dem Commit in das zentrale Repository
* **TODO**: Testautomation?