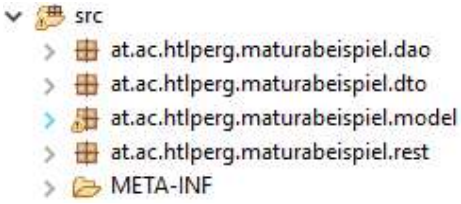
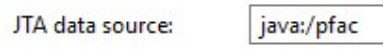
JPA – Backend

- **MySQL-Server starten** – mysqld.cmd
- **Wildfly** starten
- **JDBC Treiber** am Wildfly deployen
- **Schema** in **MySQL Workbench** erstellen
- **Datasource** im Wildfly anlegen
  - Localhost:9990
  - Configuration -> Start
  - Subsystems -> Datasources -> Non-XA -> Add
  - MySQL -> JNDI Name vergeben -> Detected Driver -> Nicht den Fabric/h2 auswählen!
  - Wichtig: Bei Connection URL statt mysqldb EUREN Schemanamen eintragen
  - Username: root, Password ist leer; Test Connection unbedingt ausführen
  - JNDI Name kopieren/merken
- **Dynamic Webproject** anlegen
  - Target Runtime = WildFly 11.0 Runtime; den Rest lassen
  - Bei web.xml Deployment Descriptor Hackerl setzen



- Rechtsklick aufs Projekt -> Convert to **JPA Project**
  - JNDI Name in persistence.xml unter Connection eintragen!!!



  - Package Model anlegen -> Rechtsklick -> New -> Other -> JPA-Entities from Table
    - Gewünschte Tabellen auswählen
    - Key generator = identity
  - In den generierten Klassen können im Normalfall OneToMany Beziehungen einfach gelöscht werden, da diese LazyInit Exceptions verursachen; Man kann diese jedoch auch im Dao mittels einer JPQL Query beseitigen → dann nicht löschen

```
return em.createQuery("SELECT b from Bill b LEFT JOIN FETCH b.lines", Bill.class).getResultList();
```

- **Dao** – ein Dao pro Entität, wenn man diese für Rest benötigt
  - Named und Scope Annotations festlegen

```
@Named
@RequestScoped
public class TicketDao {
```

  - EntityManager als PersistenceContext festlegen

```
@PersistenceContext
EntityManager em;
```

  - GetAll

```
public List<Ticket> getAll() {
    return em.createNamedQuery("Ticket.findAll", Ticket.class).getResultList();
    //return em.createQuery("Select t from Ticket t LEFT JOIN FETCH t.assignments", Ticket.class).getResultList();
}
```

  - GetOne

```
public Ticket get(int id) {
    return em.find(Ticket.class, id);
}
```

- Create → Transactional Annotation wichtig!

```
@Transactional
public void create(Ticket toInsert) {
    em.persist(toInsert);

    System.out.println(toInsert.getId());
}
```

- Update → Transactional

```
@Transactional
public void update(Ticket toUpdate) {

    Ticket ticket = get(toUpdate.getId());
    System.out.println(toUpdate.getDescription() + " " + ticket.getDescription());

    if(ticket != null) {
        ticket.setDescription(toUpdate.getDescription());
        ticket.setSubmittedOn(toUpdate.getSubmittedOn());
        ticket.setPriority(toUpdate.getPriority());
        ticket.setState(toUpdate.getState());
        ticket.setUser(toUpdate.getUser());


        em.merge(ticket);
    }

}
```

Objekt vorher vom EntityManager holen, um das ManagedObject zu erhalten und bei diesem dann die neuen Werte setzen – nicht das übergebene Mergen!

- Delete → Transactional

```
@Transactional
public void delete(int id) {
    Ticket toRemove = get(id);
    if(toRemove != null) {
        em.remove(toRemove);
    }
}
```

- **DTO**
  - Um ManyToOne Beziehung nach außen besser darzustellen
  - Client muss z.B. dann nur die ID des abhängigen Elements kennen und nicht das genaue Objekt

```
public class TicketDto {
    public int id;
    public String description;
    public Date submittedOn;

    //Priority
    public int priorityID;
    public String priorityName;

    //State
    public int stateID;
    public String stateName;

    //User
    public int userID;
    public String lastName;
    public String firstName;
    public String userName;
}
```

Attribute der abhängigen Objekte (Priority, State und User) als public Attribute in DTO einfügen

- **Rest**
  - Configure -> Add JAX-RS Support
  - Rechtsklick in Package Rest -> New JAX-RS Resource
    - Target Entities auswählen, die nach außen repräsentiert werden sollen
    - XML entfernen und Hakerl bei den gewünschten Operationen setzen
  - Benötigte Daos injecten
  - Nach außen nur DTOs schicken und von außen nur DTOs entgegennehmen

```
@POST
public Response create(final TicketDto ticket) {
    //TODO: process the given ticket
```

- DTO erzeugen/aus DTO Objekt erzeugen

```
private TicketDto createTicketDto(final Ticket t) {
    TicketDto dto = null;
    if(t != null) {
        dto = new TicketDto();
        dto.id = t.getId();
        dto.description = t.getDescription();
        dto.submittedOn = t.getSubmittedOn();
        dto.priorityID = t.getPriority().getId();
        dto.priorityName = t.getPriority().getPriorityName();
        dto.stateID = t.getState().getId();
        dto.stateName = t.getState().getStateName();
        dto.userID = t.getUser().getId();
        dto.userName = t.getUser().getUserName();
        dto.firstName = t.getUser().getFirstName();
        dto.lastName = t.getUser().getLastName();
    }
    return dto;
}
```

```
private Ticket createTicketFromDto(final TicketDto t)  {
    Ticket ticket = new Ticket();

    if(t != null) {

        final State s = stDao.get(t.stateID);
        final User u = uDao.get(t.userID);
        final Priority p = pDao.get(t.priorityID);

        if(s != null && u != null && p != null) {
            ticket.setId(t.id);
            ticket.setDescription(t.description);
            ticket.setSubmittedOn(t.submittedOn);
            ticket.setState(s);
            ticket.setPriority(p);
            ticket.setUser(u);
        }

    }

    return ticket;
}
```

- Create
  - Dto in richtiges Objekt mittels der obigen Methoden umwandeln
  - Wenn null → Bad-Request
  - Ansonsten den langen, auskommentierten Header verwenden
- ListAll
  - Returnwert: List<*EntityNameEinfügen*Dto>

```
final List<TicketDto> tickets = dao.getAll().stream().map(ticket -> createTicketDto(ticket))
                                .collect(Collectors.toList());
return tickets;
```

  - Alle Objekte in Dto umwandeln
- Update
  - Dto in richtiges Objekt umwandeln und überprüfen, ob nicht null und in DB vorhanden → OK
  - Sonst NOT_FOUND
- Delete
  - Überprüfen, ob in DB
  - OK oder NOT_FOUND

Angular Client

- Ng new projektname –routing
- Ng generate component name
- Ng generate interface IName
- Ng generate service Data
- Für Two-Way Binding FormsModule aus @angular/forms in app.module.ts
- Services in app.module.ts in providers einfügen
- Routing
  - Routen definieren

```
const routes: Routes = [
  { path: "home", component: ListComponent },
  { path: "edit/:id", component: EditComponent },
  { path: "", redirectTo: "/home", pathMatch: "full" }
];
```

- Parameter aus Route → Activated Route und paramMap.map → Observable!

```
constructor(private route: ActivatedRoute, private http: HttpClient
  private router: Router, private dataService: DataService) { }

ngOnInit() {
  this.editID = this.route.paramMap.map(param => param.get('id'));
```

- Routerevent manuell auslösen

```
this.router.navigate(["home"]);
```

- Routerevent über <a> auslösen

```
<a routerLink="/edit/{{ticket.id}}">Edit</a>
```

- Daten aus Observable in Tabelle anzeigen

```
<tr *ngFor="let ticket of ticketList | async">
  <td>{{ticket.description}}</td>
```

Mittels „async“ kann man die Daten ohne Subscribe direkt anzeigen
Zugriff auf Attribute mit {{}}

- Dropdown

```
<select (change)="setState($event.target.value)" name="stateName">
  <option [selected]="toInsert.description.length == 0">Select State</option>
  <option *ngFor="let state of stateList | async">
    {{state.stateName}}
  </option>
</select>
```

$event ist das ausgelöste Event
So kann man mittels event.preventDefault() im Code z.B. einen Pagereload vermeiden

- Element aus Observable<irgendwos[]> finden

```
this.stateList.subscribe(states => {
  const state = states.find(s => s.stateName === stateName);
  this.toInsert.stateName = stateName;
  this.toInsert.stateID = state.id;
});
```

- Array von Objekten von API abfragen
  - Angular httpclient
    - In app.module.ts HttpClientModule aus@angular/common/http
    - In Komponente HttpClient aus @angular/common/http
  -

```
return this.http.get<Ticket[]>(this.URL + "tickets/");
```

- Objekt einfügen/updaten
  - Einfügen = POST → KEINE ID ANGEBEN → am besten ID = undefined im Objekt

    `id: undefined,` in toInsert

    ```
    return this.http.post(this.URL + "tickets/", toInsert);
    ```
  - Update = PUT

    ```
    return this.http.put(this.URL + "tickets/" + toUpdate.id, toUpdate)
    ```
- Wichtig ist, für alle Entities Interfaces zu erstellen, um Types nutzen zu können
- Bei POST, PUT und DELETE muss bei den Observables subscribed werden!!!