Verontschuldigd: Tim, Peter, Bart
Aanwezig: Tommy, Broes, Johan

Release beta-1

- Uitgesteld tot einde van de vakantie. Na de vakantie sluit ik me nog eens een week op, om de opgelopen achterstand te kunnen inhlaen

Datacontracts

Het datacontract PersoonInfo hebben we ongeveer een jaar geleden geïntroduceerd, om voor een personenlijst niet telkens alle op te lijsten personen helemaal over de lijn te moeten sturen. We staken in PersoonInfo enkel de zaken die we in een algemene lijst wilden weergeven, vandaar dat er bijvoorbeeld 'VolledigeNaam' inzit; de combinatie voor- en familienaam.

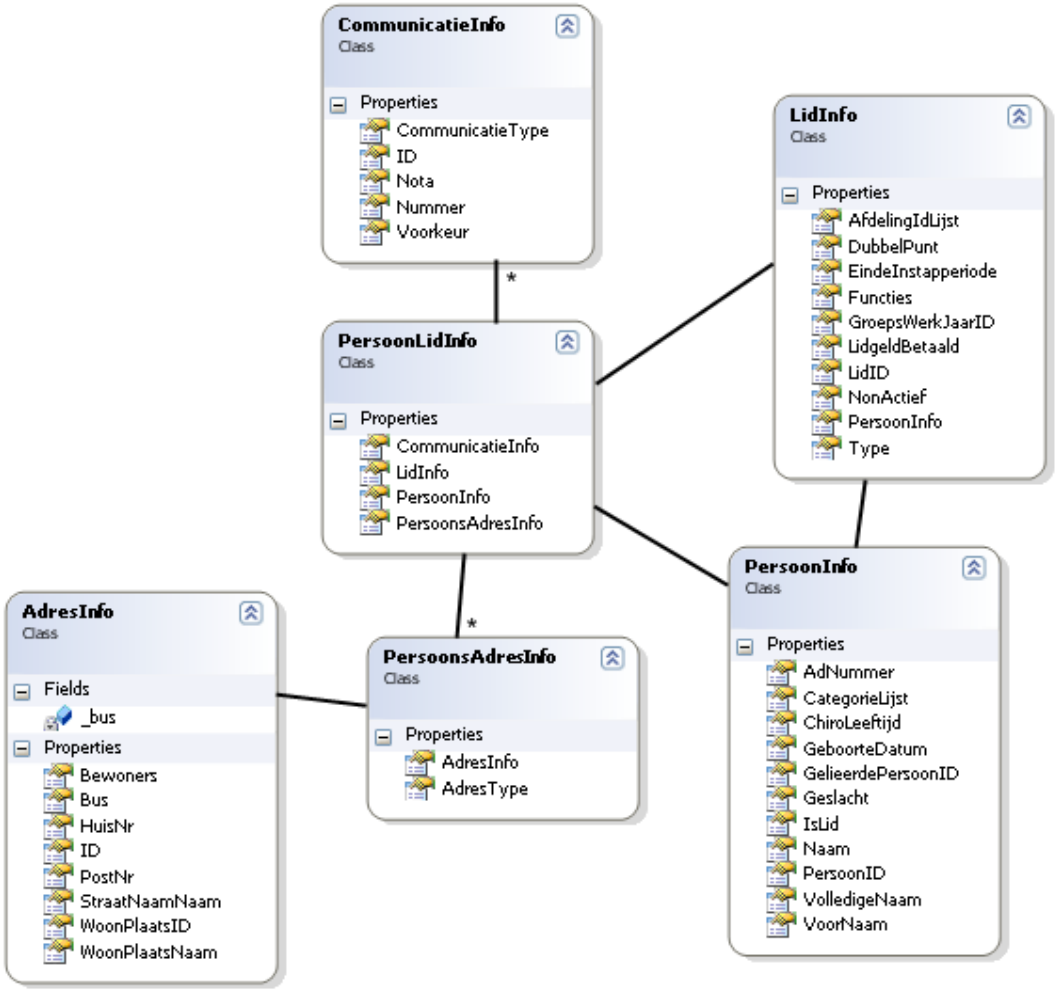
Wanneer een (gelieerde) persoon bewerkt wordt, dan wordt het model GelieerdePersonenModel gebruikt, en meer bepaald GelieerdePersonenModel.HuidigePersoon, en die is van het type 'Persoon'.

Analoog maakten we het datacontract LidInfo, voor de overzichtslijst van de leden.

Voor het bewerken van een lid gebruiken we LedenModel.HuidigLid, en die is van het type 'LidInfo'.

LidInfo is dus wel geschikt voor het bewerken van de gegevens van een lid. PersoonInfo is niet geschikt voor het bewerken van de gegevens van een persoon.

Ondertussen bestaat er ook nog een datacontract ‘PersoonLidInfo’, en ziet het volledige plaatje er als volgt uit:



~~Ik denk dat we naar volgende situatie moeten:~~

- ~~Het huidige datacontract PersoonInfo hernoemen als BeperktePersoonInfo (of zoiets)~~
- ~~Een nieuw datacontract PersoonInfo maken, ongeveer als volgt:~~

```
public class PersoonInfo
{
        public int AdNr;
        public string VoorNaam;
        public String Naam;
        // ... En alle andere persoonsgebonden members

        public IList<CategorieInfo> Categorieen;
        public IList<CommunicatieInfo> Communicatie;
        public IList<AdresInfo> Adressen;
        public LidInfo LidInfo;
}
```

~~Waarbij uit LidInfo de persoonsgebonden informatie wegvalt.~~

Conclusie:

- 'Platte' datacontracten genieten de voorkeur
- Voor lijsten kan wel gecombineerd worden, om het aantal service calls te beperken
- Datacontracten heen en datacontracten terug moeten niet per se dezelfde zijn

- We moeten een onderscheid maken tussen beperkte datacontracten (voor overzichten), en uitgebreide datacontracten (voor view/edit van 1 element). (#432)
    - Lichte datacontracts eindigen op –Info
    - Zware datacontracts op –Detail.
- De 'schrijfdatacontracten' (naar de service) lijken sterk op de entity-classes, om automapper gemakkelijk te kunnen gebruiken.
- Na refactoring willen we geen entity classes meer exposen via de service. (als data- of servicecontracts) (#443)
- In het concreet geval PersoonLidInfo: (#434)
    - Verbinding PersoonInfo – LidInfo verwijderen
    - PersoonsAdresInfo en AdresInfo combineren we.
- Situatie met 'GroepsExtras', 'PersoonsExtras',…
    - Ideaal zou zijn: voor elke vraag een aparte service method en een apart datacontract
    - Maar dat is veel werk
    - Voorlopig houden we de pragmatische oplossing met 'GroepsExtras', 'PersoonsExtras', …

Exception handling

Situatieschets

We bevinden ons in deze situatie:

- De user interface communiceert via de back-end via services. We willen het aantal service-calls beperkt houden.
- De service moet gepast reageren als er iets gevraagd wordt dat niet mag/kan volgens de businesslogica.
- De businesslaag moet waken over de businesslogica, en niet de UI/servicelaag

Deze vaststellingen hebben een effect op onze implementatie:

- Als er onjuiste zaken worden gevraagd aan de businesslaag, gooit deze een exceptie op, en wordt die doorgegeven via een servicecontract.
- Als de UI iets gedaan wil krijgen van de business, dan wordt die vraag gewoon 'gesteld' aan de service. Als er een exception terugkomt, voert de UI de juiste actie uit.

De UI moet dus voldoende informatie kunnen afleiden uit de opgeworpen exception, om te kunnen bepalen wat er precies misgelopen is.

Poging tot 'stroomlijnen' exception handling

Ik heb een aantal Custom Exceptions geïmplementeerd.

```
        [Serializable]
        public class FoutCodeException<T> : System.Exception, ISerializable
        {
            private T _foutCode = default(T);

            public FoutCodeException(T foutCode) : this(foutCode, null, null) { }

            public T FoutCode
            {
                get { return _foutCode; }
                set { _foutCode = value; }
            }

            // …
        }
```

Een FoutCodeException is een exception die een foutcode bevat, waarbij de foutcode typisch een enum is. De beschikbare foutcodes vind je in https://develop.chiro.be/trac/cg2/browser/trunk/Solution/Chiro.Gap.Fouten/FoutCodes.cs.

Een voorbeeld:

```
            if (!f.IsNationaal && f.Groep.ID != lid.GroepsWerkJaar.Groep.ID)
            {
                throw new FoutCodeException<VerkeerdeGroepFoutCode>(
                        VerkeerdeGroepFoutCode.Functie,
                        Properties.Resources.FoutieveGroepFunctie);
            }
```

- Generic type voor foutcode-enum: geen goed idee.
- Per specifieke situatie 1 exception. Eventueel wel details in de exceptieproperty's. En dat kunnen enums zijn, maar strings, ints zijn zeker even goed.
- We maken 1 'basisexceptie' voor GAP, waar al onze custom exceptions van zullen inheriten. (#435)
    - GapException
        - Int FoutNummer
        - String[] items // voor substitutie in foutboodschap

In sommige gevallen is het wenselijk om 'problematische objecten' mee op te nemen in de exception. Hiervoor heb ik de 'BlokkerendeObjectenException':

```
[Serializable]
public class BlokkerendeObjectenException<TEntiteit>
        : GapException where TEntiteit: IBasisEntiteit
{
        private IEnumerable<TEntiteit> _objecten;

        public BlokkerendeObjectenException(
                int foutCode,
                IEnumerable<TEntiteit> objecten)
               : this(foutCode, objecten, null, null) { }

        public IEnumerable<TEntiteit> Objecten
        {
                get { return _objecten; }
                set {  _objecten = value;  }
        }

         // …
}
```

Als de service nu zo'n exception 'catcht', dan moet die over de lijn via een faultcontract.

```
[DataContract]
public class BlokkerendeObjectenFault<TObject>: GapFault
{
        [DataMember]
        public IEnumerable<TObject> Objecten { get; set; }
}
```

Het volledige verhaal is nu het volgende:

- De businesslaag throwt een exception
- De servicelaag catcht de exception
- De servicelaag mapt de exception naar een faultcontract, en stuurt een faultexception over de lijn.

Voor die laatste stap gebruik ik automapper (en nu wordt het lelijk):

```
public int CategorieToevoegen(int groepID, string naam, string code)
{
        Groep g = _groepenMgr.OphalenMetCategorieen(groepID);
        try
        {
                Categorie c = _groepenMgr.CategorieToevoegen(g, naam, code);
        }
        catch (BlokkerendeObjectenException<Categorie> ex)
        {
                var fault = Mapper.Map<
                          BlokkerendeObjectenException<Categorie>,
                          BlokkerendeObjectenFault<CategorieInfo>>(ex);
                throw new FaultException<BlokkerendeObjectenFault<CategorieInfo>>(
                          fault);
        }

        // …
}
```

- Uiteraard moeten de mappings dan ook gedefinieerd worden; ~~zie~~ https://develop.chiro.be/trac/cg2/browser/trunk/Solution/Chiro.Gap.ServiceContracts.Mappers/MappingHelper.cs#L198~~.~~ (Voor Automapper: het is de gewoonte om de map te creeren net voor ze gebruikt wordt. Automapper heeft zelf een cache.)

Aan de kant van de UI kan nu gepast gereageerd worden.

```
try
{
	ServiceHelper.CallService<IGroepenService>(svc => svc.CategorieToevoegen(
		groepID,
		model.NieuweCategorie.Naam,
		model.NieuweCategorie.Code));
		return RedirectToAction("Index", new { groepID = groepID });
}
catch (FaultException<BlokkerendeObjectenFault<CategorieInfo>> ex)
{
	// …
}
```

~~Omdat die generics wel heel wat tekst met zich meebrengen in de code, kunnen eventueel 'aliassen' gemaakt worden voor veel gebruikte exceptions/faultcontracts~~
Zoals gezegd erven specifieke exceptions van GapException

```
	[Serializable]
	public class GeenGavException : GapException
	{
		public GeenGavException(
			GeenGavFoutCode foutCode, string[] items)
			: base(foutCode, items) { }
	}
```

Conclusies:

- Generiek maken om objecten mee te geven is minder een probleem. Bijv: BlokkerendeObjectenException<T>: GapException.
- Mappen van exceptions naar faultcontracts met automapper: OK
- Exception als detail in Faultcontract: geen goed idee. Dit exposet informatie naar buiten. (Zie ook wet van Demeter, die we langzaamaan aan het invoeren zijn.) (#436)

'Terug naar de lijst'

Welke soort state kunnen we gebruiken:

- Sessies (20 min timeout)
- Caching (timeout; informatie op server)
- Application (global; gaat niet)
- Database (omslachtig)
- Cookies (enkel bij browser in client)
  - Tijdelijk (blijft bestaan zolang de browser bestaat)
  - permanent
- URL query-parameters
- Hidden fields (veel werk, alle controller actions moeten volgen)

Besluit: url bewaren in **tijdelijk cookie**. Iedere keer de gebruiker een lijst bekijkt, wordt de url in de cookie aangepast. Als de cookie niet meer bestaat/leeg is, dan gaat de teruglink gewoon naar 'home' of iets gelijkaardig. (Personenlijst/Ledenlijst/… wat er van toepassing is) (#437)

! Er bestaan .NET-classes voor cookies. Die gaan we dan ook gebruiken.

Programmeerweekend

- Heibrand, van zaterdag 22 mei, 9.00 uur t/m zondag 23 mei, 15.00 uur.
  - Ik vraag na of we later kunnen blijven. (want maandag is't toch verlof ☺)
- Ik zou van iedereen willen weten wanneer hij aankomt/vertrekt
  - Tommy is er heel de tijd.

Varia

- Exceptions moeten het volgende implenteren:
  - protected CustomException(SerializationInfo info, StreamingContext context) // constructor
  - override GetObjectData
  - …
- Code checken met FxCop. (eventueel automatisch via post-build events)
  - Beginnen met alles af te zetten
  - ‘rubrieken’ een voor een activeren