Composite UI

Michał Padzik

padzikm@gmail.com

https://github.com/padzikm/CompositeUI

Preconditions:

- Each service is logically and physically responsible for its data and for performing operations on its data
- Each service can render its data in an appropriate format
- Each service is being developed independently

Guides how to start:

- Elements on a page are viewmodels, which can render itself, apart from carrying data
- So called BrandingService is responsible only for creating a grid for elements and put each one of them in a right place
- So called IT/OpsService integrates services and provides configuration

Potential integration problems:

- Central point of dependency
- Need to know what, where and when to call
- Possible need for collaboration between services (ex. in model binding, validation, naming, etc)
- Change in one service can lead to changes in other services
- Dependency injection conflicts

Solution step 1:

- Let’s name and write down each element that will be on every page
- Having named elements for a given page, BrandingService will know where to put each one of them and can order viewmodels to render themselves
- Viewmodels are delivered by integration service – how to collect them?

Solution step 2:

- Elements identified on a page – HttpContext knows everything
- Instead of calling services for a given request, let’s create request mapping mechanism – routing
- Routing is one and it is global, so let’s introduce internal routing for each service as a copy of a global one
- One change – namespaces for a specific service
- This way services that are interested in a given page can themselves deliver viewmodels
- How to render viewmodel if its views and actions are in separate routing than it is called?

Solution step 3:

- ViewModel has to know how to find a way to its resources – let’s register services in a global routing in an unique way
- Each route will contain key (ex. service name – it’s unique), which will fire only if we give this specific key
- In order to call a view from a viewmodel we passed the key, and because every nested call from a view will inherit route data, the key will be automatically copied too
- Starting from a view we’re moving in a specific service by default
- Where to get physical view file from?

Solution step 4:

- For every service let’s create a viewengine, which thanks to the key in a routing will know if a view request is addressed for it
- Many repetitive implementations depending on the key – let’s add a bit of intuitive conventions and let T4 generates it for us
- Internal service components want to communicate with each other (ex via ajax) – how to get directly to a specific destination without polluting main application?

Solution step 5:

- Let’s register services in a global routing as areas with unique prefix (ex. service name) and let’s reserve areas names in a global application that match given pattern (ex. with Service suffix)
- This give us unique, direct incoming routes for services – for controllers in main folder we define area: NameService/controller/action and for any proper area we define: NameService/area/controller/action
- How to pass data from client and main application to services?

Solution step 6:

- Again HttpContext knows the answer

- Every service knows what it put, so it also knows what and how to get it back

- One process allows sharing session, cookies, etc between services, which can be used to leave breadcrumbs to each other (ex. main app creates Id for a given resource and each service has access to it)
- What about transactions?

Solution step 7:

- One process allows us to enlist all/some operations in one transaction/many transactions
- Distributed transaction by the definition, but ex. by using msmq communication will be only inside local machine
- What about dependency injection?

Solution step 8:

- Dependency graph has a single entry point – controller, which is unique for every service
- Into main container let’s pass dependencies required for steering application (ex. routing, viewengines, controlleractivator, etc) and for each service let’s create separate dependency container
- As responsibilities of main application and services are different and separate, there is no conflict between containers

Solution step 9:

DEMO