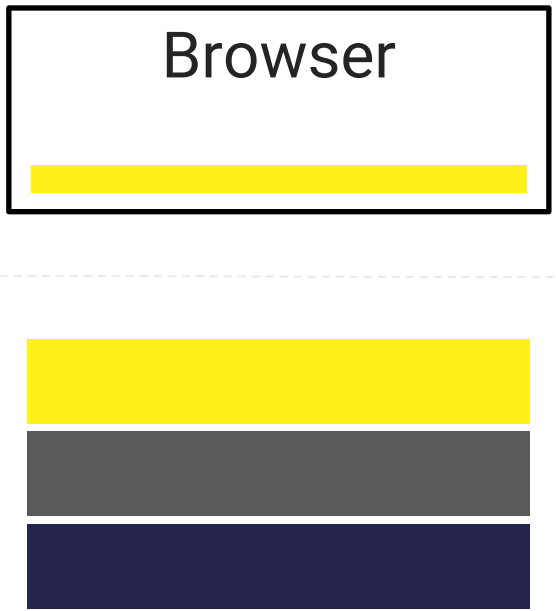
Container Bootcamp

UI Modularization

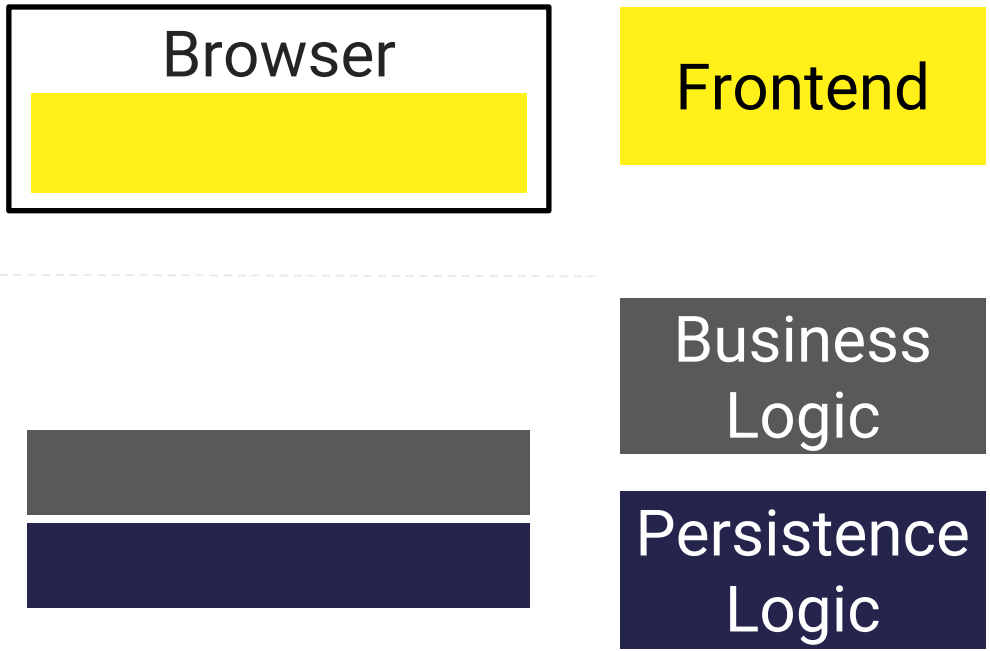
Web Application Styles

Classic Web Applications



- Classic Web Application
- Renders HTML
- Plus some JavaScript enhancements

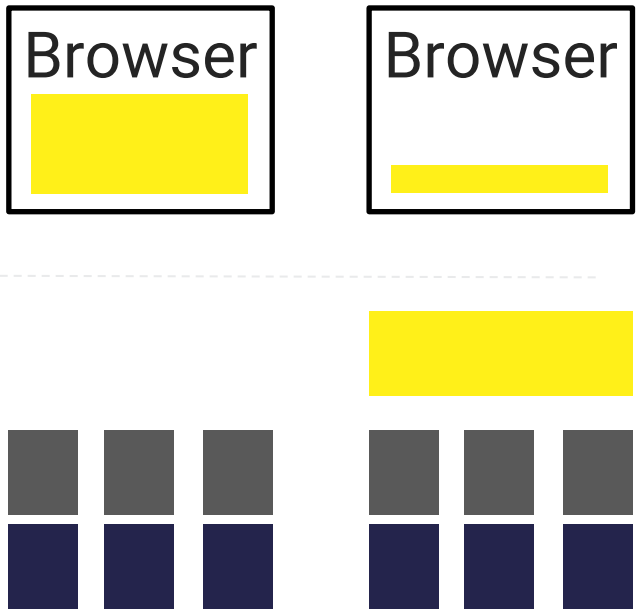
Single Page Apps



- Single Page App
- All frontend logic in JavaScript
- Server provides e.g. JSON / REST

UI Integration

Integration -- UI Monolith



- Logic and persistence modularized
- UI Layer one monolith
- Single Page App
- Classic Web App
- Mobile App

UI Monolith: Why?

- Separate UI team builds its own component
- One team = one component

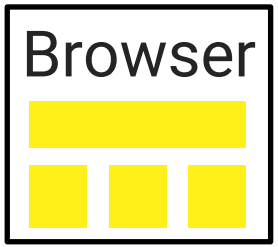
- Mobile app: monolithic by nature

UI Monolith: Advantages & Disadvantages

- Easy to build
- Uniform look & feel not too complex

- Changes go through two teams
- ...and at least two components
- Changes need multiple deployments

Integration Options -- Modularized UI Monolith



- Separated modules
- ...in a Single Page App
- ...or Mobile App

Modularized UI Monolith: Why?

- SPA and Web App are monolithic by nature
- Need to separate development

Advantages & Disadvantages

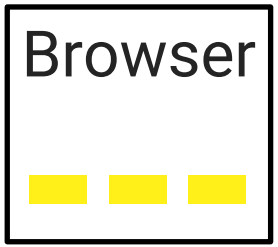
- Better than a pure monolith
- Separate development
- ...but no separate deployment
- Might need to coordinate development closely

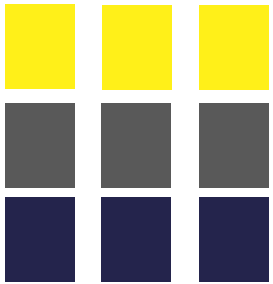
Modularized UI -- Implementation

- AngularJS, Ember, …
- Integration via framework facilities e.g. AngularJS Modules
- Similar to classic modules
- Shared code / code dependencies

Why build backend modules if your UI is a monolith?

Integration Options -- Separate Deployment





- Each module has its own UI module
- Separately deployable
- Probably no common assets
- …because they are code dependencies

Separate Deployment: Why?

- Need to deploy and build features in one team
- …independently from all others
- i.e. one team can build one feature

Advantages & Disadvantages

- Completely independent development and deployment
- No runtime dependencies on the server
- Progressive enhancement possible
- Graceful degradation possible
- But: High technical complexity
- Common look & feel hard

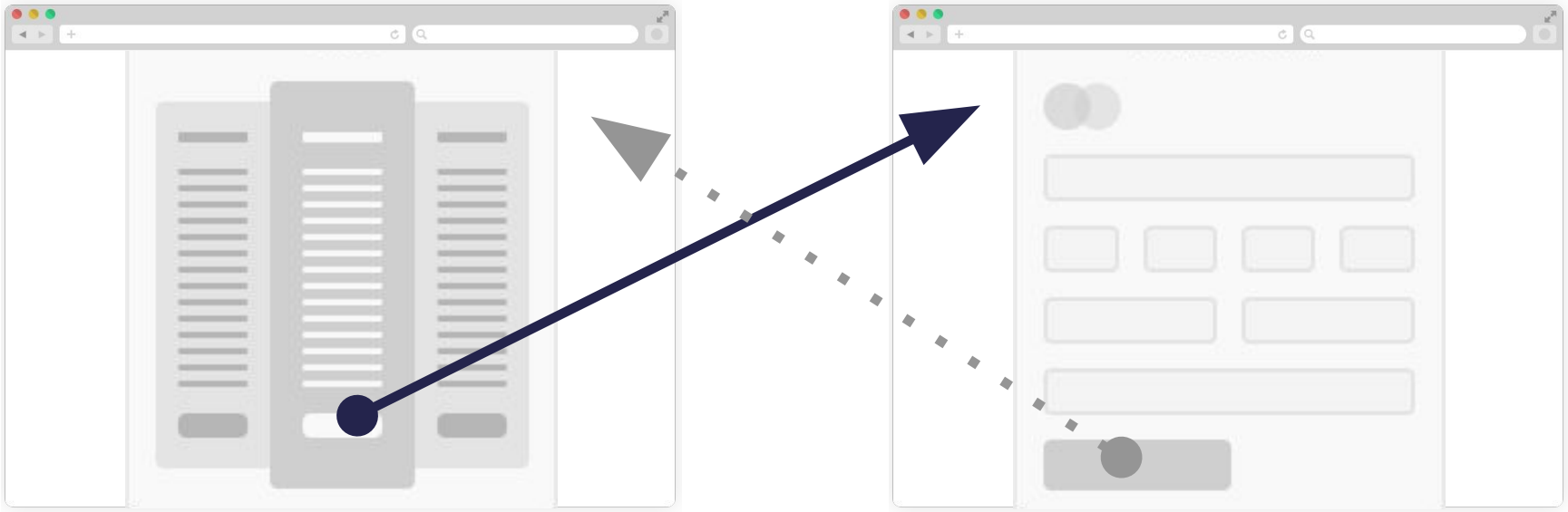
Links

System 1

System 2

Hyperlinks to navigate between systems.

Redirection

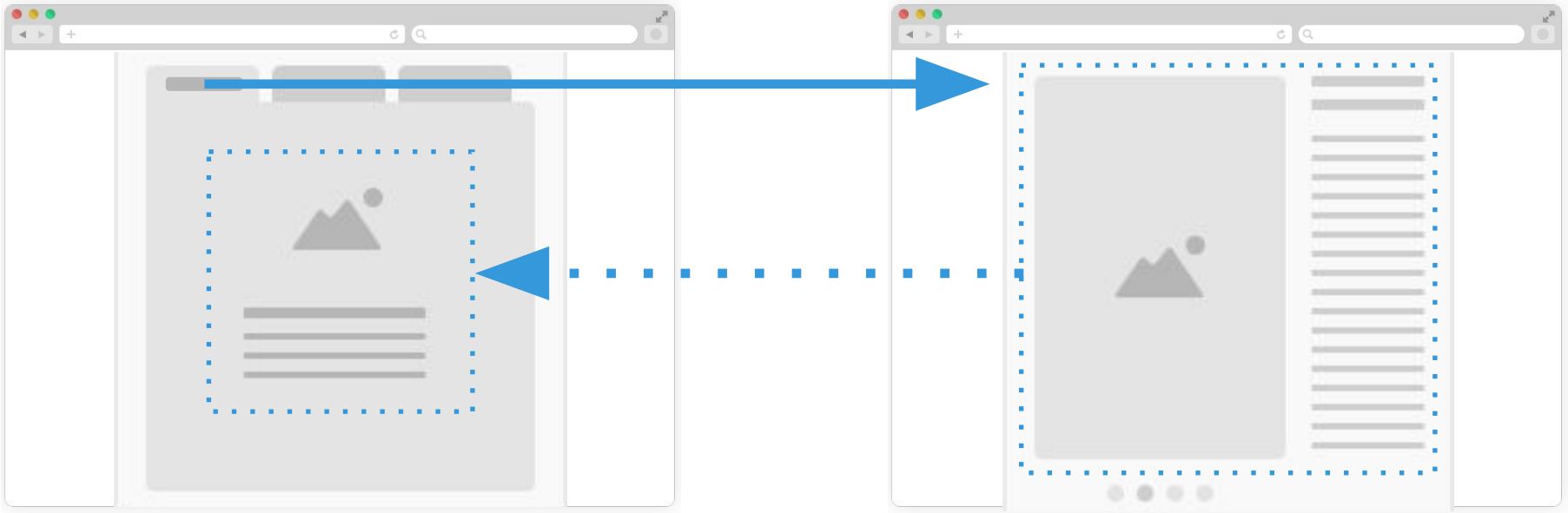


System 1

System 2

- Use of callback URIs
- As seen e.g. in OAuth flows

Transclusion

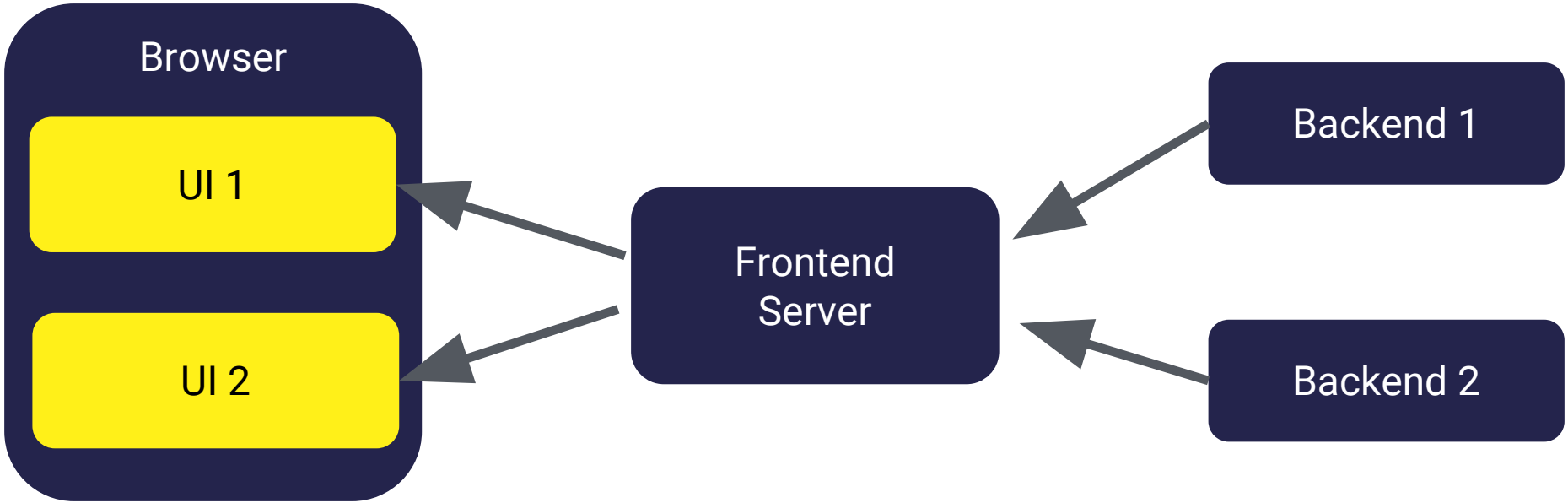


System 1

System 2

Dynamic inclusion of content served by another application

Server-side integration



- **Centralized aggregation**
- **Bottleneck (runtime/development time)**

ESI (Edge Side Includes)

```
...
<header>
... Logged in as: Ada Lovelace  ...
</header>
...
<div>
  ... a lot of content and images ...
</div>
...
<div>
  Footer stuff
</div>
```

ESI (Edge Side Includes)

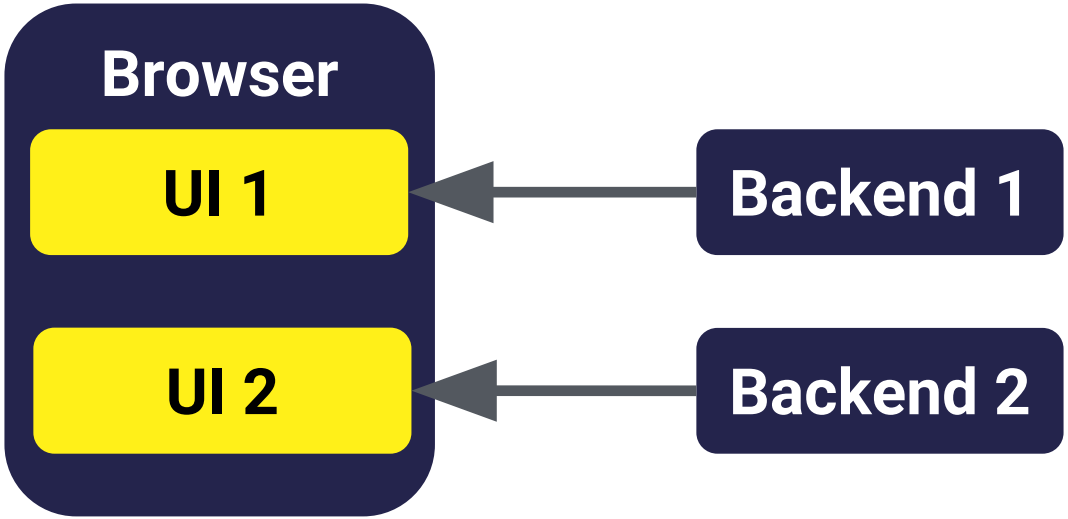
```
...
<esi:include src="http://example.com/header" />
...
<div>
  ... a lot of content and images ...
</div>
...

<esi:include src="http://example.com/footer" />
```

Similar approaches

- SSI (Server-side includes) (Apache, Nginx)
- Portal server: Just a modularized frontend, no integration of several backends
- Homegrown solutions

Client-side integration



- Proprietary integration
- Client requirements (e.g. CORS, JS)
- Upcoming: HMTL Imports

What to use when?

- SSI: Often no additional software needed
- ESI: More resilience and performance due to caching
- Page should be usable without JavaScript
- i.e. use ESI/SSI for fundamental parts
- Client-side to beautify

What to use when?

- Client-side can reduce load on server
- Client-side can reduce time to first render

Conclusion

- UI Integration for very loose coupling
- SSI: Server-side on standard software
- ESI: Server-side + caching + resilience
- Client side e.g. jQuery