AEM React

AEM components written in React

History

date	author	message	changed chapters
21.3.17	Stefan Meyer	- Using AemRoute to Router doc (#42) - Added documentation for including vanilla react components and other AEM components. - improved Router documentation	• /Getting Started/First component • /Development Guide/ResourceComponent • /Development Guide/Registering react component • /Development Guide/Loading Resource • /Development Guide/Java Api • /Development Guide/Vanilla component • /Configuration/Javascript • /Single Page Application/Example
4.3.17	stefan meyer	added transitions - added transitions - fixed rendering of wrapper element in partial updates - migrated to typings	• /Development Guide/Vanilla component
1.3.17	stefan meyer	added notes to archetype usage about updating dependencies	• /Getting Started/First project

1 Introduction

The goal of the AEM (Adobe Experience Manager) React library is to use React as a templating engine for AEM components. React is a popular javascript ui library by facebook.

Why React and AEM?

React components are ideal to create web applications with complex client-side interactivity. AEM provides a perfect authoring interface for web content. This project brings these technologies together, so that you can build highly interactive web pages with a professional authoring tool.

Features

- Universal React rendering
- High performance javascript execution with a pool of Java 8 nashorn engines.
- Nesting React components in other AEM components and vice versa is supported.
- Converting vanilla (plain) react components into AEM components is supported.
- SPA based on react router is supported.

Projects

AEM project

The aem-react project consists of the following parts:

- osgi bundle contains the Sling Script Engine to render AEM components written in react.
- maven archetype is a fork of the AEM archetype and adds react support and examples.
- demo content package provides examples for components and SPA.

Maven artifact is available via maven central

Javascript project

The aem-react projects relies on the aem-react-js subproject, which provides the basic javascript funtionality. It is available as npm module.

Maven archetype project

Lastly a maven-archetype is avilable to quickly create an AEM maven project including react components. It is a fork of the existing offical AEM maven archetype.

Maven artifact is available via maven central

> **Version**
> This documentation always describes the latest version (not release) of these projects.

Prerequisites

- >= Java 8 (Oracle JDK with nashorn engine)
- >= AEM 6.0

2 Getting Started

In this guide we will use the maven archetype to generate a project structure. It already includes some demo content and react components.

Requirements

- Adobe Experience Manager 6 or higher
- Apache Maven (3.x should do)

The example react components are written in Typescript. The node build tools are also assuming that you are developing in Typescript. If you want to use another dialect you need to tewak the build scripts.

2.1 First project

To quickly get started we will use the maven archetype.

1. create maven project

```
mvn archetype:generate \
 -DarchetypeGroupId=com.sinnerschrader.aem.react \
 -DarchetypeArtifactId=aem-project-archetype \
 -DarchetypeVersion=10.x \
```

> **Versions**
>
> Get the latest archetypeVersion. After generation of the project make sure that you also have the latest versions of the aem-react-js npm module in ui.apps/src/main/ts/package.json and the latest aem-react osgi bundle in core/pom.xml.

You will then be asked a couple of questions about project name and folder names and so on. These are the same as in the original archetype. Please find detailed explanations here in the section "Getting started in 5 minutes".

Available properties

groupId	Maven GroupId
groupId	Base Maven groupId
artifactId	Base Maven ArtifactId
version	Version
package	Java Source Package
appsFolderName	/apps folder name
artifactName	Maven Project Name
componentGroupName	AEM component group name
contentFolderName	/content folder name
cssId	prefix used in generated css
packageGroup	Content Package Group name
siteName	AEM site name

2. start AEM

AEM should now be running.

3. install demo

```
mvn install -PautoInstallPackage
```

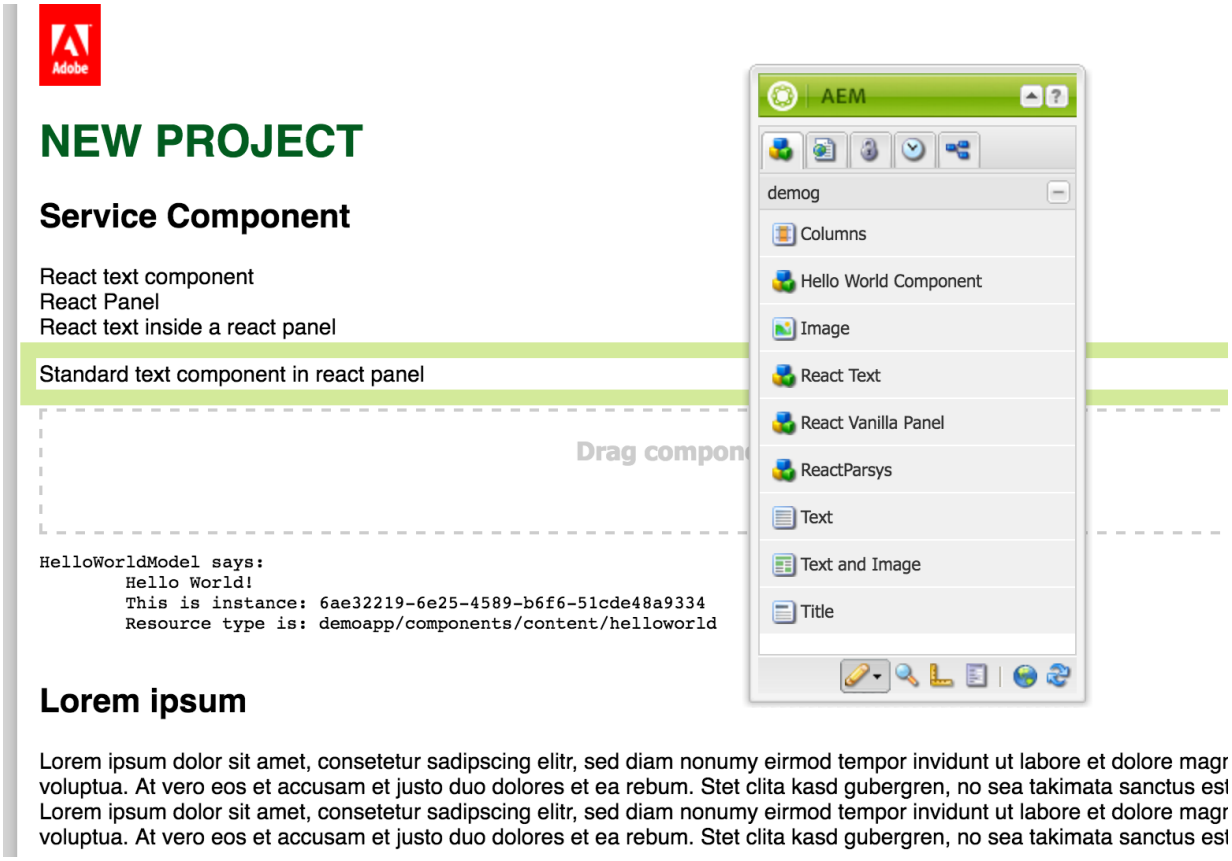
If your AEM instance is not running on localhost:4503 then you need to add additonal parameter

parameter	default
-Daem.port	4502
-Daem.host	localhost

4. Open browser

To check what was deployed we will use the Classic UI.

- go to the page /content/${siteName}/en.html
- find react components in sidekick: *React Panel*, *React Text* and *ReactParsys*
- find components already on the page

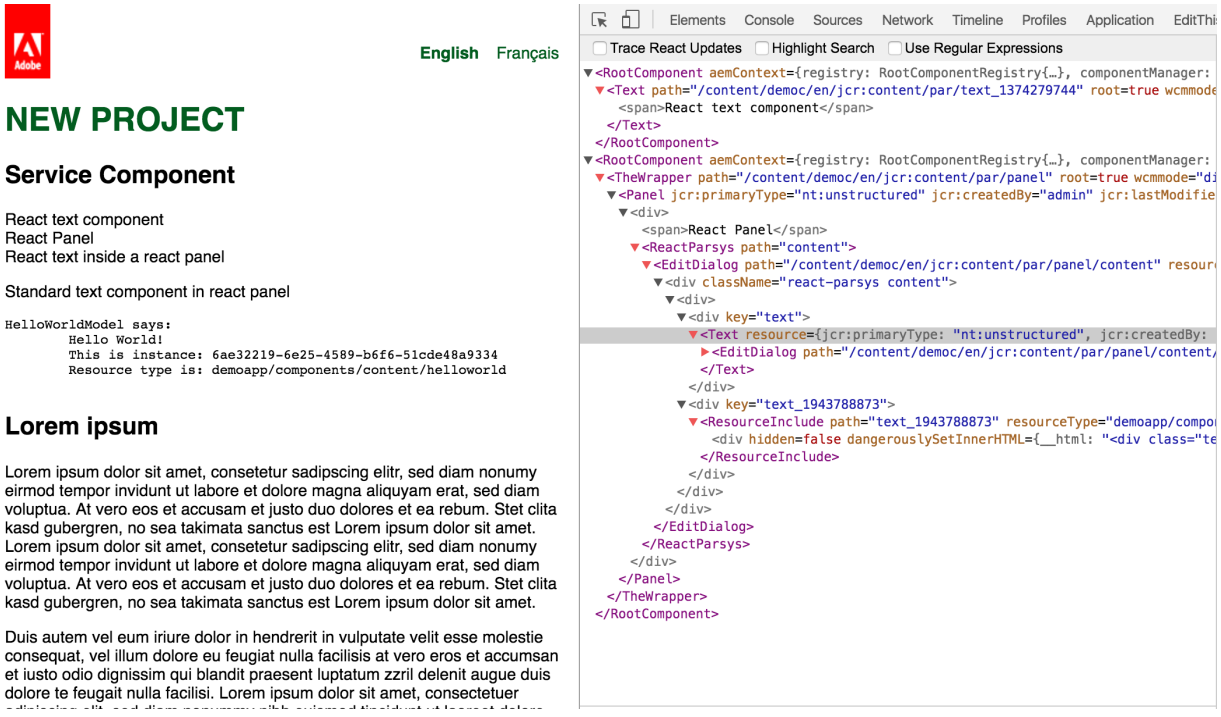


The page en.html with sidekick

5. disable author mode

To check that we have actual react components in the page we will use the Classic UI.

- Disabled the author mode by appending `?wcmmode=disabled` to the url.
- Install your react dev tool in chrome browser.
- Have a look at react component tree.



The page en.html with react dev tools

There are two *RootComponents* which means two independent react component trees. The first contains the *Text* component. The second contains the *Panel* which is a vanilla react component and therefore wrapped by a *TheWrapper* component. The Panel's child is a *ReactParsys* which contains another *Text* component and a non-react component which inserted by the *ResourceInclude* component.

2.2 First component

This part assumes that the project was created according to the previous chapter.

1. Start watch task

Start the watch task which transpiles, bundles and uploads the javascript files to AEM.

Open console to folder /src/main/ts and run the watch task `npm run watch`.

If your AEM instance is not running on localhost:4502 then you need to make these configurations:

```
npm config set demo:crx http://admin:admin@localhost:4502/crx/repository/crx.default
```

Alternatively the config in the package.json can be modified.

2. create file

Create a file MyComponent.tsx under /ui.apps/src/main/ts/.

````typescript jsx

import {ResourceComponent} from "aem-react-js/component/ResourceComponent"; import * as React from "react"; import Text from "../text/text";

export default class MyComponent extends ResourceComponent { public renderBody(): React.ReactElement {

```
    let label: string = this.getResource().label;
    return (

            Hello {label}
            <Text path="text"/>

    );
}
```

}

```
# 3. Register component

The component needs to be associated with a resourceType `${appsFolderName}`/components/my-component.
Open /ui.apps/src/main/ts/componentRegistry.tsx and add two lines

````typescript jsx
// add this line at the top
import MyComponent from "./MyComponent";
...
// add this line after componentRegistry is instantiated
componentRegistry.register(MyComponent);
```

### 4. Create component configuration

Create the component configuration in the appropriate folder `/apps/${appsFolderName}/components/my-component`. The template is an empty file called `my-component.jsx`. The edit dialog should provide a textfield for the property `./label`.

- /apps/${appsFolderName}/components/my-component
  - .content.xml
  - my-component.jsx
  - dialog.xml

### 5. Synchronize source code to crx

The component configuration must be uploaded to crx. This can be done via maven install -PautoInstallPackage. The watch task has already uploaded the javascript file.

### 6. Open browser

find the new react component in the sidekick.

### 7. Continuely improve component

# 2.3 Demo

The aem-react project contains a demo content package.

**To run the demo**

1. Clone git repo

   ```
   git clone https://github.com/sinnerschrader/aem-react.git
   ```

2. Install into running AEM

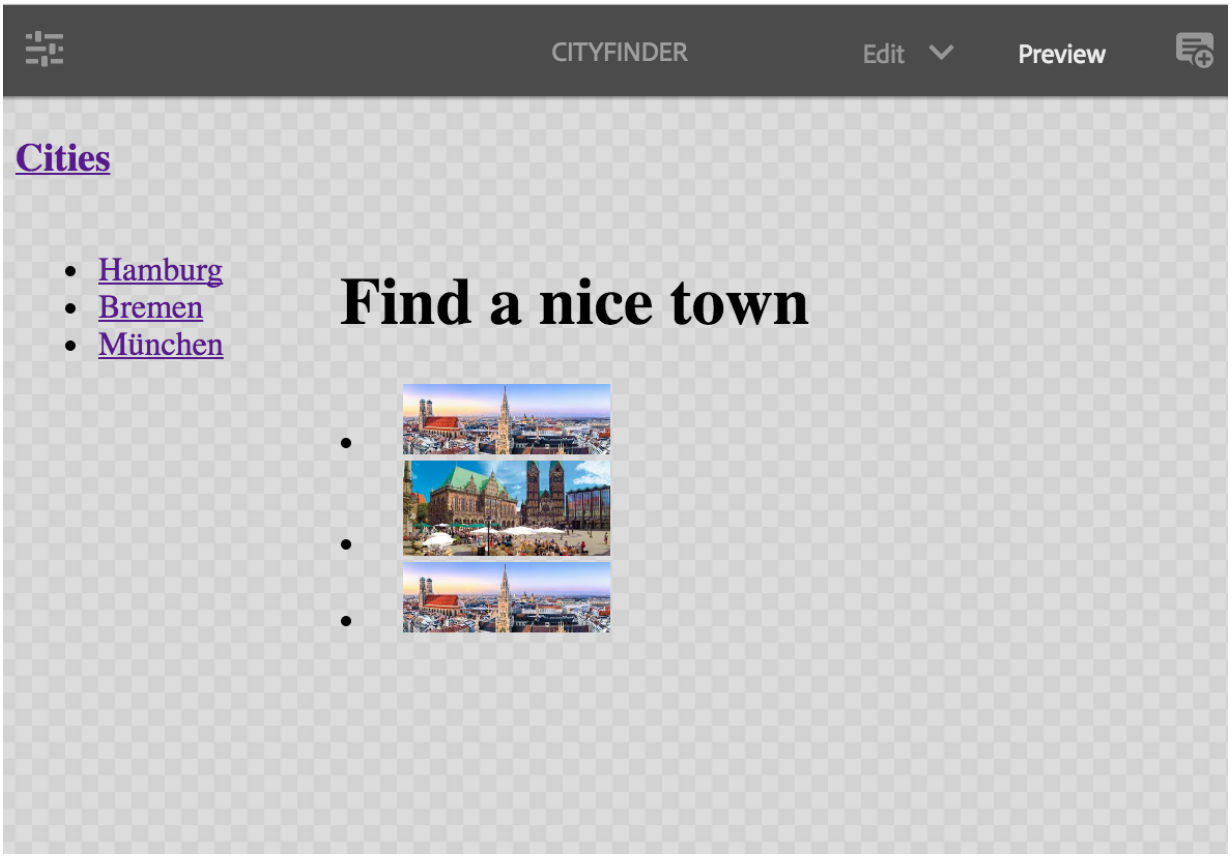
   ```
   mvn install -PautoInstallPackage -Daem.port=<port> -Daem.host=<host>
   ```

> Currently it is not possible to run the demo and the archetype in the same AEM. You need to adjust the path to the javascript in the react script engine via the webconsole.

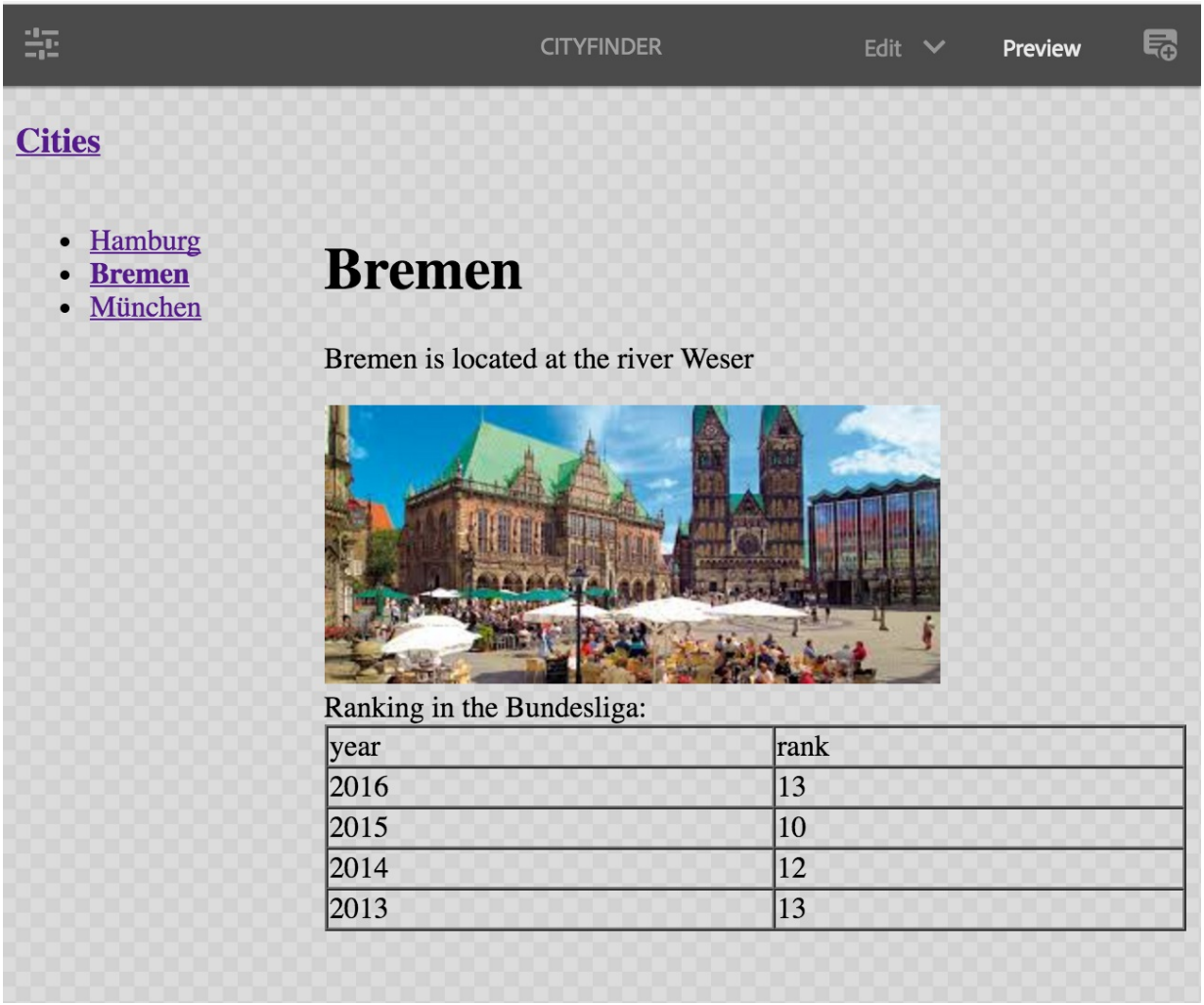
**SPA with a two level page hierachy**

Local url: /content/reactdemo/cities.html

The SPA (single page application) is based on the react router library. The SPA has a welcome view (/cities.html) and a detail view for each city (e.g.: /cities/hamburg.html).




*Welcome view*




*Detail view*

The main AEM react component is `CityFinder`. A city can be selected from the list on the left. This list is part of the `CityListView` react component which is displayed on all views. Its child component is either `Home` for the welcome view or `CityView` for the detail view. In the author mode each view is a single AEM page. Each page contains the CityFinder component which contains the router. `Home` and `CityListView` are plain react components while CityFinder and CityView are AEM react components.

A simplified version of `CityFinder` looks like this:

```
export default class CityFinder {
  renderBody() {
    return (

      <Router history={history}>
        <Route path="cities.html" component={CityListView}>
          <IndexRoute component={Home} baseResourcePath={resourcePath}/>
          <Route path="cities/(:name).html" resourceComponent={CityView} component={ResourceRoute}/>
        </Route>
      </Router>

    );
  }
}
```

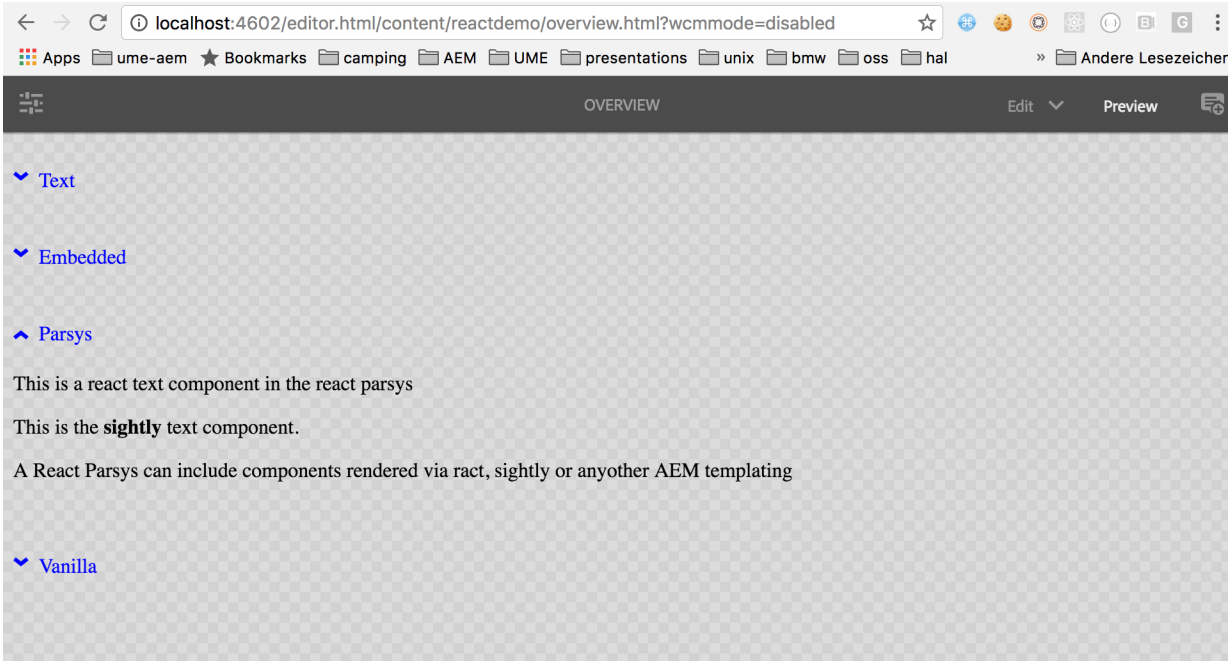
The purpose of the generic `ResourceRoute` component is to translate the current routes dynamic path into a resource path which is passed to `CityView`.

**Component list**

Local url: /content/reactdemo/overview.html

The list of components include:

- simple text component
- embedded component
- accordion component which shows client-side interactivity and serves as a container.
- vanilla component
- vanilla panel which serves as a container




*Overview of example components*

# 3 Development Guide

## 3.1 ResourceComponent

ResourceComponent is the base class for AEM components. It provides access to the resource (content). It also adds the necessary wrapper element, so that the component can be edited in the author mode.

The main method to implement in a ResourceComponent is `renderBody()`. It is called by the `render` method once the resource is successfully fetched. The resource is available via `this.getResource()`.

```typescript jsx
import {ResourceComponent} from "aem-react-js/component/ResourceComponent"; import * as React from "react"; import Text from "../text/text";

export default class MyComponent extends ResourceComponent { public renderBody(): React.ReactElement {

    let label: string = this.getResource().label;
    return (
        <div>
            <span>Hello {label}</span>
        </div>
    );
}
```

}

```
# Container

If the component is a container then it must to render its children by itself. One option is to call `renderChildren` which
turns the component into a parsys like container for any child components. If the child components are restricted to a certain type
 then a custom rendering might be a better solution.


```typescript jsx

public renderBody(): React.ReactElement<any> {

    let label: string = this.getResource().label;
    let children: React.ReactElement<any>[] =this.renderChildren(this.getResource(), "children");
    return (
        <div>
            <span>Hello {label}</span>
            {children}
        </div>
    );
}
```

### Embed AEM components

To embed another AEm component you use `<ResourceInclude/>` and pass path and resourceType.

```typescript jsx
return (
)
```
```

## 3.2 Registering react component

There must be one instance of `RootComponentRegistry`. It is responsible for mapping each React component to a resourceType. A component is a registered with one of the `ComponentRegistry`s which are maped to a resource path.

````typescript jsx import ComponentRegistry from "aem-react-js/ComponentRegistry"; import RootComponentRegistry from "aem-react-js/RootComponentRegistry"; import MyComponent from "./MyComponent";

let registry: ComponentRegistry = new ComponentRegistry("yourproject/components"); registry.register(MyComponent); // resource type of MyComponent is 'yourproject/components/my-component'

let rootComponentRegistry: RootComponentRegistry = new RootComponentRegistry(); rootComponentRegistry.add(componentRegistry); rootComponentRegistry.init(); AemGlobal.registry = rootComponentRegistry; // expose registry to Nashorn

````

If your project was created by the maven archetpye then the RootComponentRegistry is already instantiated and you just need to add your `ComponentRegistry`.`

3.3 Loading Resource

The resource will be loaded as json by calling `getResource()` and therefore the number of levels of the resource tree need to be defined in advance by overriding the method `getDepth()`. In accordance with the sling conventions 0 means a single level.

````typescript jsx public getDepth(): number { return 2; }

```
# Lazy Loading

If the resource is not fetched synchronuously then the `render` method will call `renderLoading` instead to display a loading spinner or
similar ui.
Asynchronuous loading happens when a ResourceComponent's path prop is changed or a new ResourceComponent
is added to the resource tree in the client. This is often the case when the react router library is used.

````typescript jsx
  public renderLoading(): React.ReactElement<any> {

    return (
      <span>Loading data ...</spany>
    );
  }
```

## 3.4 Author mode

React components are not instantiated in the author mode to prevent anyy interference between AEM javascript and react. For a lot of components this means that they need to be displayed differently. For example an Accordion must display all panels and its corresponding parsys. Use `isWcmEnabled()` on the server to detect the author mode.

## 3.5 Java Api

Presentation logic is often implemented in sling models. To access a sling model or an osgi service the fully qualified java class name needs to be passed to appropriate method. The object returned is a `aem-react-js/ServiceProxy`, which has a single method `invoke`. That method's first parameter is the actual java method to invoke and the remaining parameters are passed to that method.

| method | description |
| --- | --- |
| getResourceModel(className) | adapt the current resource to the given class name. |
| getRequestModel(className) | adapt the current request to the given class name. |
| getOsgiService(className) | get the osgi service by its service class name |

`typescript jsx import ServiceProxy from "aem-react-js/ServiceProxy"; ... public renderBody(): React.ReactElement<any> { let model: ServiceProxy = this.getRequestModel(`com.example.LabelModel`); let label: string = model.invoke('getLabel') return ( <div> <span>Hello {label}</span> </div> ); }`

The Java API methods will be invoked only if the component's is rendered on the server. Otherwise the return value is served from the cache which was created during server rendering. It is safe to invoke a java method in the `renderBody` method. But it must be invoked unconditionally and always with the same parameters. Initially `renderBody` will always be invoked on the server but it can be invoked on the client many times after that.

If a service is needed to load data based on user input then you should not use the Java api but use a custom http service via plain ajax.

## 3.6 Vanilla component

A vanilla react component can be registered as a AEM component as well.

````typescript jsx registry.registerVanilla({component: TextField}); registry.registerVanilla({component: Panel, parsys: {path: "content"}, depth: 2});
````

All resource properties are passed as props to the component.

For a simple component that only needs a single level of the resource tree
and doesn't display children it is sufficient to define the React component class
that should be registered. The following additional parameters are available

parameter	type	description
depth	number	the number of levels of the resource available
props?	any	extra props that are passed
parsys?	object	define this property to define a parsys as the only child of this component
parsys.path	string	the relative content path to store children
parsys.className?	string	class name added to the parsys element
parsys.elementName?	string	name of the parsys element (default is "div")
parsys.childElementName?| string| If provided each child is wrapped in an extra element
parsys.childClassName?| string| class name added to children elements.
transform?	function	a function to tranform the resource into react props

 # Container

If the `parsys` property is set then the vanilla component will be turned into
an AEM container. The `.content.xml` must also set the corresponding attribute:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<jcr:root xmlns:sling="http://sling.apache.org/jcr/sling/1.0" xmlns:cq="http://www.day.com/jcr/cq/1.0" xmlns:jcr="http://www.jcp.org/jcr/1.0"
    jcr:primaryType="cq:Component"
    jcr:title="My Title"
    cq:isContainer="true"
    />
```

Resource transformation

If the resource's structure does not match the props of the vanilla component then a transformation can be used. A transformation is a function that is passed the resource and the resourceComponent and returns the props that will be passed to the react component.

In this example a sling model is used in the transformation: ```typescript jsx let transform: any = (resource: any, wrapper: ResourceComponent) => { let model: ServiceProxy = r.getResourceModel("demop.core.models.MyModel"); let newProps: any = {title: resource.label}; newProps["imageSrc"] = model.invoke("getImageSrc"); return newProps; };

registry.registerVanilla({ component: myComponent, transform: transform });

 # Include vanilla wrapper

When including a vanilla component registered as an AEM component directly in a jsx you need to use `<VanillaInclude/>`.
Otherwise it will not be editable on the page.

```typescript jsx
<div>
  <VanillaInclude path="test" component={MyVanillaComponent}/>
</div>
```

alternatively you can also use the standard include:

`typescript jsx <div> <ResourceInclude path="test" resourceType="/components/my-vanilla"/> </div>`

4 Configuration

The configuration consists of the osgi service configuration and the javascript configuration. The latter is done separately for server and client as there is a single javascript file for both server and client each.

4.1 OSGI

The main OSGI service is the ReactScriptEngineFactory which has the following properties:

name	description
scripts.paths	resource paths to the javascript files for the server
pool.total.size	pool size for nashorn engines. Correlates with the number of concurrent requests
scripts.reload	whether changes to javascript file should be observed
subServiceName	subService name for accessing the crx. If left blank then the deprecated admin is used.

4.2 Javascript

There must be two separated javascript bootstrap files. Apart from the Bootstrap files the bundled javascript file includes all the React components, which are the same for both client and server.

Server

For the server the bootstrap file must provide the method `renderReactComponent` and the `RootComponentRegistry` on the global variable `AemGlobal`. The global variable `AemGlobal` is created by the ScriptEngine. The Scriptengine will call `AemGlobal.renderReactComponent` when an AEM component is

rendered.

````typescript jsx declare var Cqx: any; declare var AemGlobal: any;

let rootComponentRegistry: RootComponentRegistry = new RootComponentRegistry(); AemGlobal.registry = rootComponentRegistry;

AemGlobal.renderReactComponent = function (path: string, resourceType: string, wcmmode: string): any { ... }

The implementation of `renderReactComponent` instantiates the the Sling implementation for the server which
uses the global variable Cqx provided **by** the ScriptEngine. Cqx is specific **to** the current request.
The configuration of the javascript is based **on** a container which must at least contain the **cache** and the sling implementation.

```typescript jsx
  let container: Container = new Container();
  container.register("javaSling", Cqx.sling);
  let cache: Cache = new Cache();
  let serverSling = new ServerSling(cache, container.get("javaSling"));
  container.register("sling", serverSling);
  container.register("cache", cache);

  let serverRenderer: ServerRenderer = new ServerRenderer(rootComponentRegistry, container);
  return serverRenderer.renderReactComponent(path, resourceType, wcmmode);
```

### Client

The javascript for the client is included in the html in the usual way. The bootstrap code must instantiate the `ComponentManager` and call `initReactComponents` on it. This should be done after the document was rendered. Both server and client have an instance of `RootComponentRegistry`, which is basically the same. One main difference between the setups is the `Sling` instance which is registered with the container.The `Sling` instance for the client uses the cache created on the server or gets data via ajax while the server instance uses the Java API directly.

````typescript jsx

interface MyWindow { AemGlobal: any; } declare var window: MyWindow; if (typeof window === "undefined") { throw "this is not the browser"; }

let rootComponentRegistry: RootComponentRegistry = new RootComponentRegistry(); rootComponentRegistry.add(componentRegistry); rootComponentRegistry.init();

let container: Container = new Container(); let cache: Cache = new Cache(); let clientSling: ClientSling = new ClientSling(cache, host); container.register("sling", clientSling); container.register("cache", cache);

let componentManager: ComponentManager = new ComponentManager(rootComponentRegistry, container); componentManager.initReactComponents();

window.AemGlobal = {componentManager: componentManager}; ````

# 5 Single Page Application

This section explains how to create a single page application (SPA) using the react-router library. The goal of a single page application is to provide content to the user spread across different views that can be navigated like any website but does not require a slow browser reload. Each view in this SPA should be bookmarkable if appropriate. The initial view should be rendered on the server for performance reasons and also to make the page crawlable by search bots and the likes.

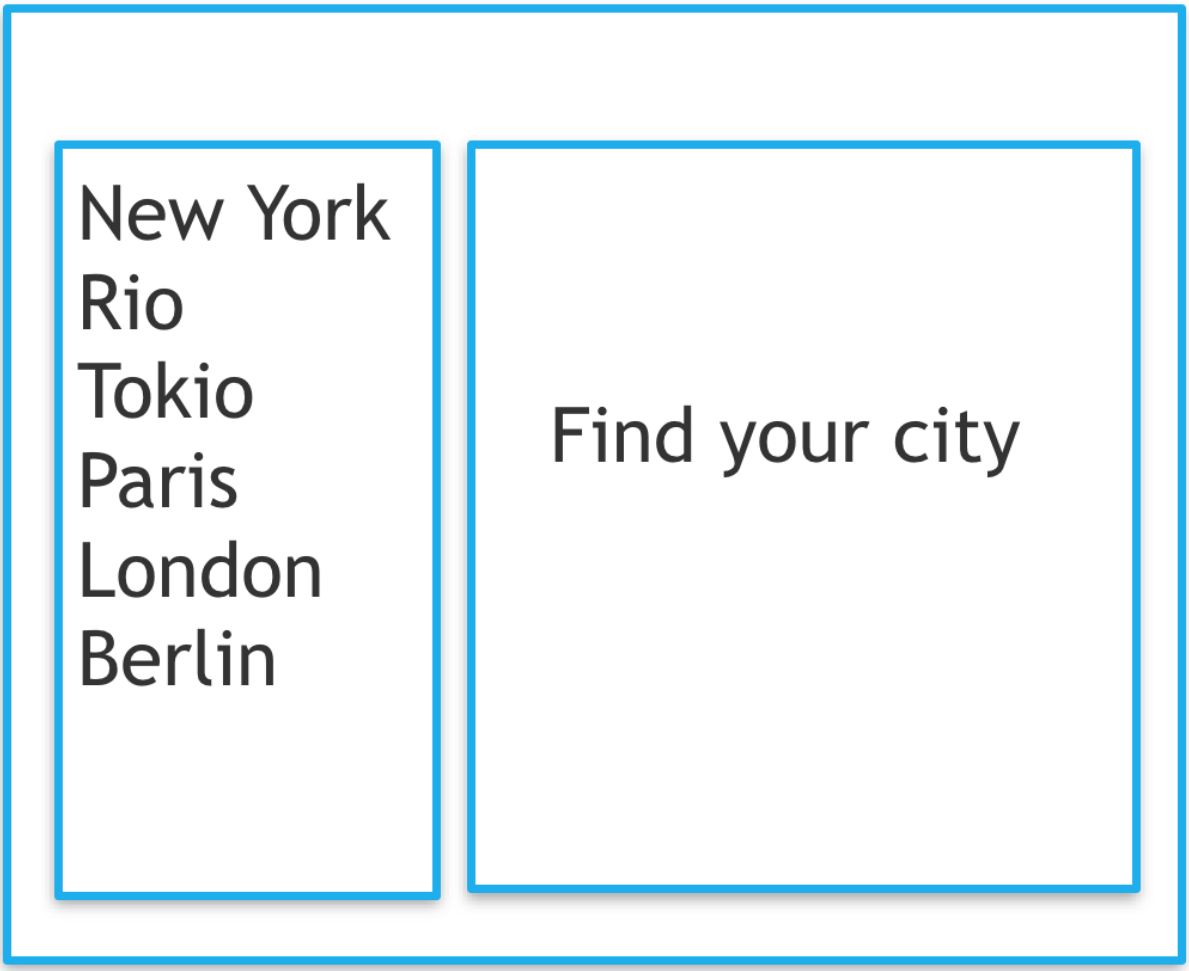
## 5.1 Example

Usually a single react component contains the router configuration. In this configuration all views and their urls are configured. These urls may contain dynamic parts, which are sometimes made available to the components comprising the views.

The example application consists of two views.

#### Welcome view

The welcome view presents a list of cities on the left and a welcome message on the right.



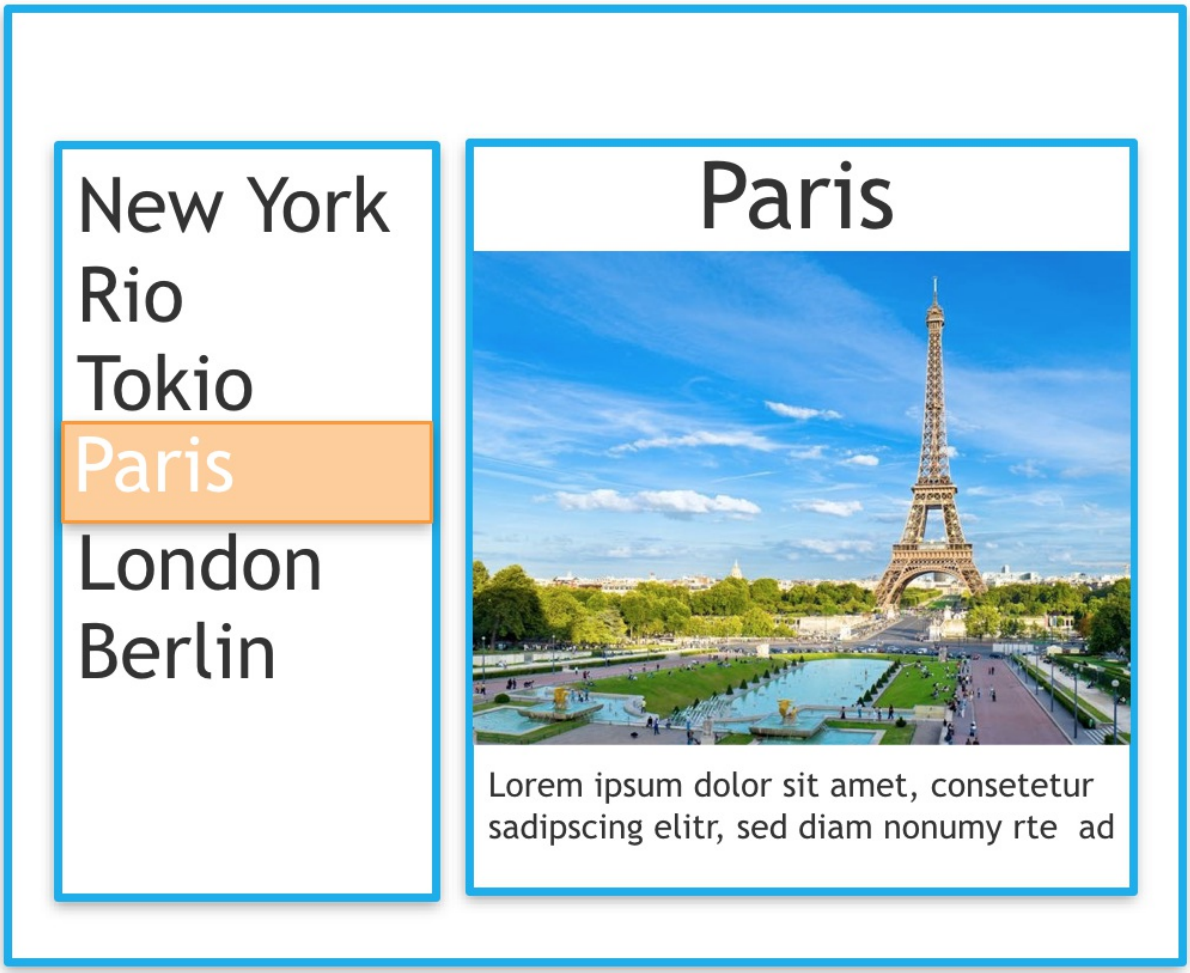

*wireframe of welcome view*

The `welcome` component tree consists of the `CityListView` and the `Welcome` component


*component tree of welcome view*

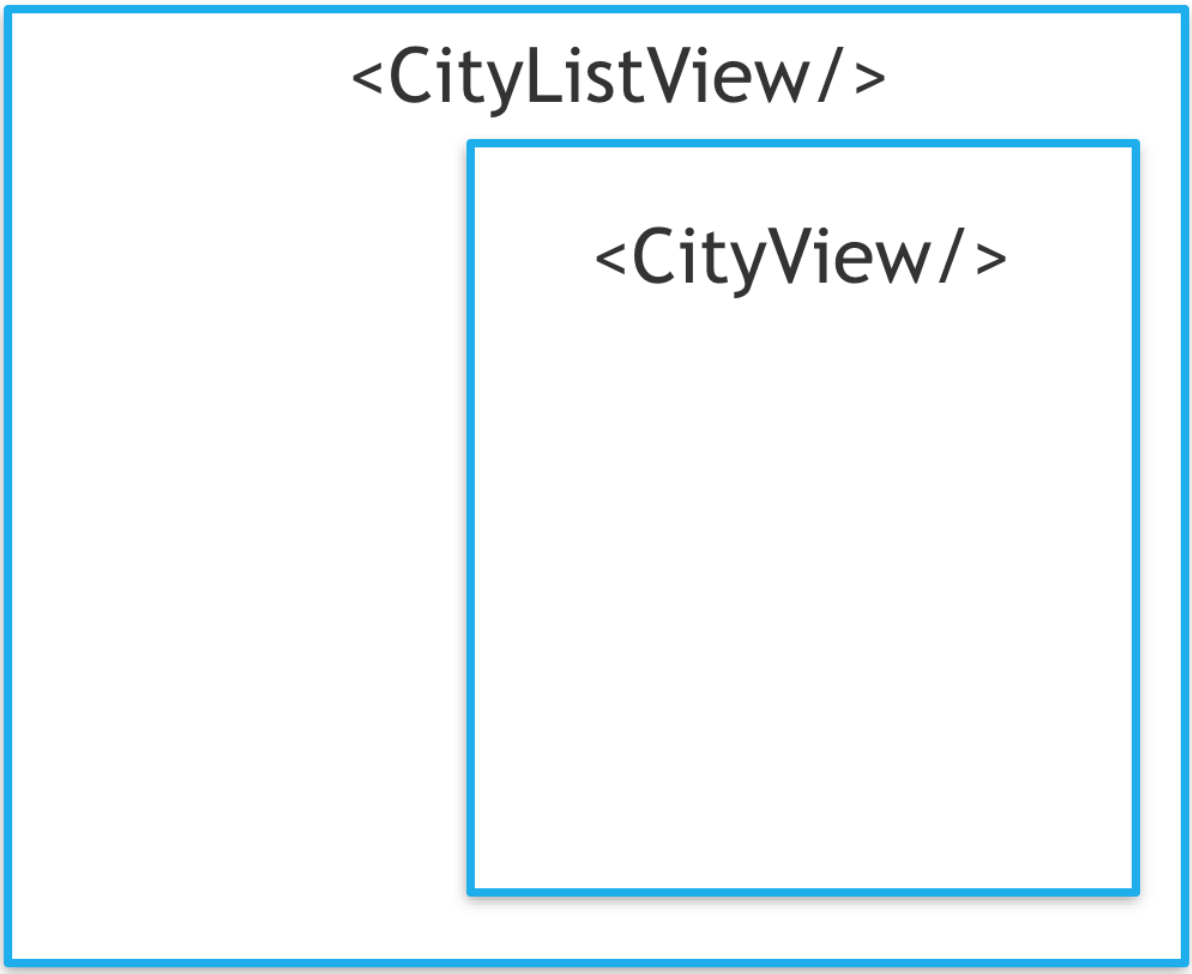
**City view**

The `city` view presents a list of cities on the left and a welcome message on the right.




*wireframe of city view*

The welcome component tree consists of the `CityListView` and the `CityView`.




*component tree of city view*

**Router**

There are two routes in this example:

1. The route to the welcome view is static: `/cities/index.html`
2. The second route points to the individual city views and is dynamic: `/cities/(:name).html`

Another aspect of the route definition is how it affects the component tree. The `CityListView` is present in all routes while its child depends on the individual route. The router configuration looks like this:

```
<Router history={history}>
  <Route path='/cities.html' component={CityListView}>
    <IndexRoute component={Welcome}/>
    <AemRoute path='/cities/(:name).html' resourceComponent={CityView} component={ResourceRoute}/>
  </Route>
</Router>
```

Note that the component of the city view is not the `CityView` component but a general `ResourceRoute` component. Its purpose is to translate the path to a resourcePath and pass it to the component defined by the `resourceComponent` property.

To use this component you create AEM pages resemble the router configuration.

- /cities.html
  - /cities/hamburg.html
  - /cities/münchen.html

> **Important**
> The router component must be located in the same path relative to the page's root (e.g. jcr:content/par/city_finder).

To decouple the absolute path from the router component it makes sense to store a depth property in the router component which makes it possible for the router component to derive the root route from the current page path.

**Example** The current page is `/cities/hamburg.html` and the depth is 1 then the root route is `/cities` and the `typscript jsx let root = ... <Router history={history}> <Route path={root+".html"}' component={CityListView}> <IndexRoute component={Welcome}/> <AemRoute path={root+"/(:name).html"} resourceComponent={CityView} component={ResourceRoute}/> </Route> </Router>`

# 6 Tools

This section describes the development tools for react components that are installed in projects created with the maven archetype. The tools are installed in

the directory `ui.apps/src/main/ts`.

## 6.1 Build Task

The build task transpiles all typescript files and bundles them into two javascript files and copies them into the appropriate target folder in the `ui.apps` project `ui.apps/target/classes/etc/designs/${appsFolderName}/clientlib-site/`:

- reactClient.js
- reactServer.js

```
npm run start
```

## 6.2 Watch Task

The watch task will watch files in the target folder of the ui.apps project and deploys them in to AEM. Start the watch task with `npm run watch`. To configure the address of the running AEM instance you need to use npm:

```
npm config set demo:crx http://admin:admin@localhost:4502/crx/repository/crx.default
```

All typescript files are watched and automatically as described in the previous chapter to two javascript files. All files below `ui.apps/target/classes/etc/designs/${appsFolderName}` are watched and deployed into AEM. This includes both the transpiled javascript files.