jQuery Ajax
POST / PUT / DELETE

1 Ajax POST/PUT/DELETE

Die Ajax-Request für GET wurden ja bereits besprochen. Die anderen Verbs – POST, PUT und DELETE – werden über die Funktion **`$.ajax()`** zur Verfügung gestellt. Letztendlich liegt auch **`$.getJSON()`** intern diese Methode auf.

1.1 .ajax()

Die vielen Parameter dieser Methode findet man unter http://api.jquery.com/jQuery.ajax/.

1.1.1 Optionen

Die wichtigsten seien hier angeführt:

url	Ziel des Ajax-Calls
method	GET, POST, PUT, …
contentType	In welcher Form werden Daten im body gesendet
dataType	In welcher Form werden die Daten erwartet
data	Daten, die im body migeschickt werden. Diese müssen mit JSON.stringify() in einen String umgewandelt werden.
success	Funktion, die nach Erhalt der Daten aufgerufen wird. Die Daten werden wie üblich als Parameter der Funktion übergeben
error	Funktion, die im Fehlerfall aufgerufen wird.
complete	Funktion, die am Ende des Requests aufgerufen wird.
beforeSend	Funktion, die vor dem Ajax-Call aufgerufen wird.

In den folgenden Methoden wird als Objekt immer folgende Variable bzw. DTO am Backend verwendet:

```
const data = {
  intVal: 666,
  strVal: 'abc'
};
```

```
public class PostData
{
  public int IntVal { get; set; }
  public string StrVal { get; set; } = "";
}
```

1.1.2 GET

Ein GET-Request mit **`getJSON`** sieht bekanntlich so aus:

```
$.getJSON(`${baseUrl}/get`)
  .then(x => lblResponse.html(JSON.stringify(x)))
  .fail(x => lblFail.html(JSON.stringify(x)));
```

Derselbe Aufruf mit der Funktion **`ajax`** sieht aus:

```
$.ajax({
    url: `${baseUrl}/get`,
    method: 'GET',
  })
  .then(x => lblResponse.html(JSON.stringify(x)))
  .fail(x => lblFail.html(JSON.stringify(x)));
```

Würde man die Callbacks als Optionen angeben, könnte man es auch so schreiben:

```
$.ajax({
    url: `${baseUrl}/get`,
    method: 'GET',
    success: x => lblResponse.html(JSON.stringify(x)),
    error: x => lblFail.html(JSON.stringify(x)),
    beforeSend: _ => console.log('Starting request'),
    complete: _ => console.log('Request handled')
  });
```

Über **`beforeSend`**/**`complete`** könnte man z.B. einen "Loading..."-Dialog zu Beginn einblenden, und nach Erhalt der Daten (bzw. im Fehlerfall) wieder ausblenden.

1.1.3 POST

Bei einem POST werden die Daten nicht als Querystring der URL angehängt, sondern als Javascript-Objekt dem Request mitgegeben. Daher müssen die Optionen **`data`** und **`contentType`** gesetzt werden:

```
$.ajax({
    url: `${baseUrl}/postFromBody`,
    method: 'POST',
    contentType: 'application/json',
    data: JSON.stringify(data)
  })
  .then(x => lblResponse.html(JSON.stringify(x)))
  .fail(x => lblFail.html(JSON.stringify(x)));
```

Man beachte, dass die Konventionen für Großschreibung in C# und Kleinschreibung in Javascript automatisch eingehalten werden. D.h. die großgeschriebenen Properties werden ohne weitere Einstellung in **kleingeschriebene** Properties des JSON-Objekts umgewandelt.
Zur Sicherheit die URL am Backend:

```
[HttpPost] public ActionResult<string> PostFromBody([FromBody] PostData postData)
```

1.1.4 PUT

Analog zu POST muss dann ein PUT-Requsest notiert werden:

```
$.ajax({
    url: `${baseUrl}/putFromBody/666`,
    method: 'PUT',
    contentType: 'application/json',
    data: JSON.stringify(data)
  })
  .then(x => lblResponse.html(JSON.stringify(x)))
  .fail(x => lblFail.html(JSON.stringify(x)));
```

Auch hier wieder die Route des Endpoints:

```
[HttpPut("{id}")] public ActionResult<string> PutFromBody(int id, [FromBody] PostData postData)
```

1.1.5 DELETE

Bei DELETE werden üblicherweise keine Daten mitgegeben, sondern nur die ID des Objekts, das gelöscht werden soll, als Teil der URL angegeben:

```
$.ajax({
    url: `${baseUrl}/DeleteSimple/666`,
    method: 'DELETE',
  })
  .then(x => lblResponse.html(JSON.stringify(x)))
  .fail(x => lblFail.html(JSON.stringify(x)));
```

Signatur am Backend:

```
[HttpDelete("{id}")] public ActionResult<string> DeleteSimple(int id)
```

1.2 .ajaxSetup()

Mit denselben Optionen wie für **`$.ajax()`** kann man mit **`$.ajaxSetup()`** Einstellungen vornehmen, die für alle weiteren Ajax-Requests gelten. Sie können aber nach wie vor bei den einzelnen $.ajax()-Aufrufen überschrieben werden.
So könnte man z.B. die Aufrufe durch folgenden Code vereinfachen:

```
$.ajaxSetup({
  contentType: 'application/json',
  error: x => lblFail.html(JSON.stringify(x)),
  beforeSend: _ => {
    lblResponse.empty();
    lblFail.empty();
    console.log('Starting request');
  },
  complete: _ => console.log('Request handled')
});
```

Der POST von üben würde sich dann entsprechend verkürzen:

```
$.ajax({
    url: `${baseUrl}/postFromBody`,
    method: 'POST',
    data: JSON.stringify(data)
  })
  .then(x => lblResponse.html(JSON.stringify(x)));
```

1.3 Monkey patch

Für derartige Aufgabenstellung bietet sich ein bei Javascript beschriebener „monkey patch“ an. Das ist wie eine Extensionmethode in C# zu sehen. Man erweitert also einfach die „Klasse“ **`jQuery`** bzw. den Alias **`$`** um eine statische post-Methode (die gäbe es zwar schon, macht aber nicht ganz das Gewünschte):

```
$.post = (url, data) => $.ajax({
  url: url,
  method: 'POST',
  contentType: 'application/json',
  data: JSON.stringify(data)
});
```

Damit wird die Verwendung dann sehr einfach:

```
$.post(`${baseUrl}/postSimple`, data)
  .then(x => lblResponse.html(JSON.stringify(x)));
```