ISEngineering

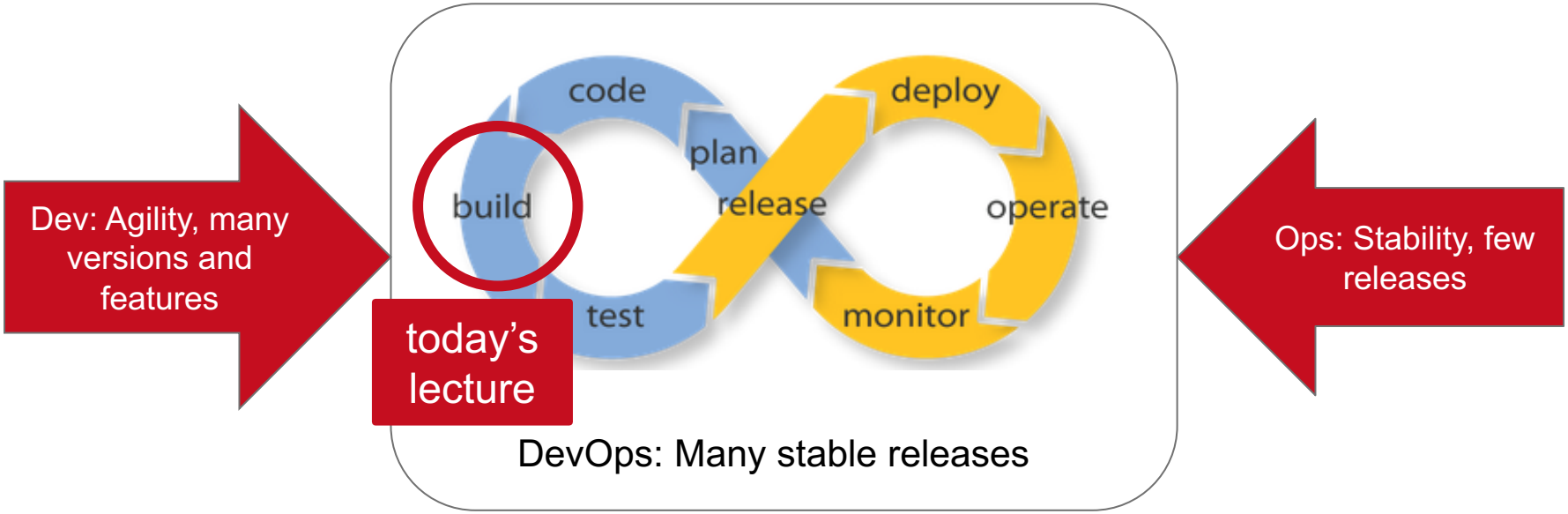
Technische Universität Berlin

CSEM: Build Automation

Dr. Markus Klems, Prof. Stefan Tai

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Dev and Ops


Dev: Agility, many versions and features
code
plan
build
release
deploy
operate
test
monitor
today's lecture
Ops: Stability, few releases
DevOps: Many stable releases

Learning objectives

- Build automation
  - Task and build automation with npm
  - Task and build automation with gulp
  - Using Docker
- Building modular applications
  - Module specifications
  - Module bundler & loader
  - Netflix A/B testing use case

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Technische Universität Berlin

BUILD AUTOMATION

Build automation

Technische Universität Berlin

Build automation is primarily about

- automating dependency (package) management
- automating tasks

Package management

- A package manager is a tool that helps users to download, install, update, and configure software packages.
  - A package is a group of multiple files that a software (program, application, library, …) consists of.
  - Packaging is simply a way of organizing software which consists of multiple files, making it easier to understand, exchange, install, or remove such a collection of files as a group (as a package).
  - A package is typically exchanged as a compressed archive that contains software (e.g., binary files, source code, configuration files) as well as meta-data for package versioning and dependency resolution.

Package management

- The local package manager (on your laptop) retrieves software from a local and/or remote repository (a special-purpose database/datastore).
- There are two main types of package managers:
  - operating system package managers (*e.g.,* yum, dpkg, or aptitude) and
  - programming language or framework package managers (*e.g.,* rubygems, pip, npm, bower).

Package management with npm

- The most popular package managers in the JavaScript universe are bower and npm.
  - Front-end JavaScript developers traditionally use bower.
  - Node.js (back-end) developers use npm.
- However, there is a trend to use npm as a universal package management tool for front- and back-end development, and even as a replacement of more complex scripting and task automation tools.

npm commands: npm init

- You can turn your directory into a node package/project by simply adding a *package.json* file.
  - With a *package.json* file, you can document the dependencies of your project and make the build and other tasks reproducible by others.
  - Generate an initial *package.json* file: `$ npm init -y`

package.json

```
{
 "name": "test-app",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 ...
}
```

npm commands: npm install

- You can add node modules as dependencies to your package/project via the npm command line with `npm install` (or short `npm i`). For example, install the “commander” module like this:

```
$ npm
npm i commander
test-app@1.0.0 /path/to/test-app
└─┬ commander@2.9.0
  └── graceful-readlink@1.0.1
```

- What did just happen? Two modules have been downloaded from the public npm registry and unpacked into your local *node_modules* subdirectory.
- The “commander” module depends on another module: “graceful-readlink”.
- The code of the “commander” module is in a single *index.js* file in the *node_modules/commander* subdirectory, which requires the “graceful-readlink” module, which consists of a single *index.js* file in the *node_modules/graceful-readlink*.

npm commands: npm install --save

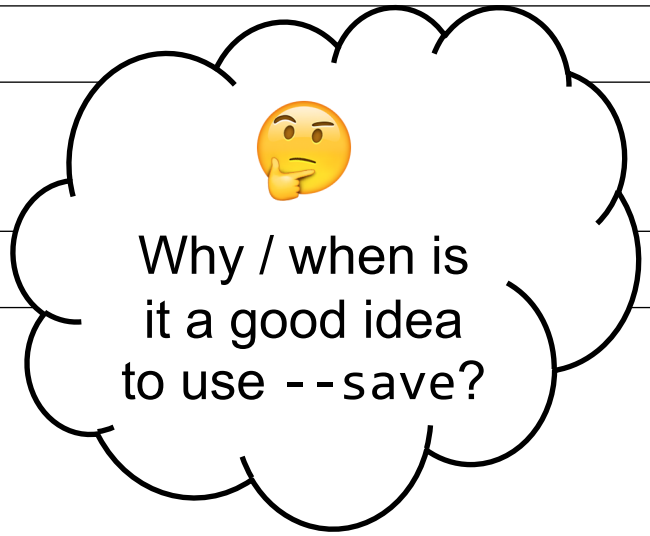
- You can add the `--save` parameter to your installation command, thereby automatically adding the dependency property to your *package.json* file.

```
$ npm
npm i --save commander
```

- The *package.json* file now looks like this:

package.json

```
{
  ... as before ...
  "dependencies": {
    "commander": "^2.9.0"
  }
}
```



npm commands: npm install --save-dev

- For distinguishing between node modules that you use in development vs. production, you can add the parameter `--save-dev` instead of `--save` to your command:

```
$ npm
npm i --save-dev mocha
```

package.json

```
{
 ... as before ...
 "dependencies": {
   "commander": "^2.9.0"
 },
 "devDependencies": {
   "mocha": "^3.0.2"
 }
}
```

Testing dependencies, such as “mocha” are only needed during development

```
$ npm
npm i --production
```

Semantic package versioning

- If you publish a node package, the initial version number should be 1.0.0. The three digits have the following meaning:
  - Patches: you update the third digit, e.g., 1.0.1 or 1.0.3. Typically used for bugfixes.
  - Minor releases: you update the second digit, e.g., 1.1.0. Typically used for minor new features which are backwards-compatible.
  - Major releases: you update the first digit, e.g., 2.0.0. Typically used for (potentially breaking, i.e., not backwards-compatible) major new features.

Semantic package versioning

- Let's say the most recent release version of the commander package is 2.9.0. Then you could use npm's semantic package versioning as follows:
  - "2.9.0" means you want exactly package 2.9.0
  - "~2.9.0" means that any patch version is fine, such as 2.9.0 or a future version 2.9.1 (however, not 2.10.0)
  - "^2.9.0" means that any minor release is fine, such as 2.9.0 or 2.12.0
  - "*" means that any version is fine, e.g., 3.0.0, 4.2.1, etc.

Task automation

- Software development is not only about writing source code. In a typical software development project, you will likely need to perform other tasks as well:
  - compile or transpile source code,
  - run unit tests,
  - start a local web server for manual testing,
  - copy and rename files,
  - minify static assets,
  - bundle your modules,
  - etc.

Task automation

- These tasks are often simple, boring, and repetitive, yet prone to manual mistakes.
- In order to improve efficiency and repeatability of your build process, there is clearly a benefit of automating these task.
- The most popular task automation tools for JavaScript development are Grunt and its successor Gulp.
- There are also other task automation tools (*e.g.,* Fly) and some developers even prefer to just use npm as a task automation tool.
- In the following, we will take a look at task automation with npm scripts and (more complex) task automation with Gulp.

Task automation with npm scripts

- The package.json "scripts" property enables you to define custom npm scripts, i.e., commands that can be executed via `npm run`.
- For example, add the following dummy "foo" property to your *package.json* file, which prints the string “bar” when you execute it:

package.json

```
{
  ... other properties ...
   "scripts": {
        "foo": "echo \"bar\""
    },
  ... other properties ...
}
```

```
$ npm
npm run foo
> test-app@1.0.0 foo /path/to/test-app
> echo "bar"
bar
```

Task automation with Gulp

Technische Universität Berlin

Gulp is an extensible and stream-oriented task runner and build automation system:

- With Gulp, you write tasks in JavaScript code instead of stating them as declarative configurations (as in Grunt).
  - This makes Gulp easy to use by JavaScript developers.
  - The flip side is that you create more code with Gulp because you are more verbose.
- Streams are central to Gulp and a unique design decision that sets it apart from other task automation tools.
  - Gulp tasks have a stream I/O interface.
  - With Gulp comes an ecosystem of plugins that can be “piped” one after the other to create a chain of stream data transformations.

Getting started with Gulp

- To get started with Gulp, you need to
  - install Gulp, for example via `npm i --save-dev gulp`, and then
  - create a *gulpfile.js*, the main configuration file in which you write the tasks that you want to automate.

Gulp API: tasks

- A gulp task can be defined like this:

gulpfile.js

```
const gulp = require('gulp');

gulp.task('some_task_name', function() {
   // do something
});
```

- First, you need to import gulp as a CommonJS-style Node.js module.
- Then you can perform the `gulp.task()` method which takes at least 2 parameters:
  - a task name and
  - a JavaScript function (the task).

Gulp API: tasks

- You can also create dependencies between tasks, e.g., first perform `'clean'` and `'foo'` tasks before you perform the `'compile'` task.
- If a task has dependencies, their task names are stated in an array as a second (optional) parameter of the task method.

gulpfile.js

```
const gulp = require('gulp');

gulp.task('compile', ['clean','foo'], function() {
   // do something
});
```

Gulp API: reading / writing files

- Gulp offers two methods for reading, and writing files, respectively:
  - `gulp.src(globs[, options])` reads data from one or more files (using glob pattern matching, e.g., * wildcards) and returns the data as vinyl file stream. The stream(s) can then be piped to Gulp plugins.
  - `gulp.dest(path[, options])` for writing data streams into a folder at the given path as files.

Gulp API: piping streams

- Gulp makes use of the Node.js `pipe()` method to create a chain of functions that are applied on a stream of data.

gulpfile.js

```
const gulp = require('gulp');
const imagemin = require('gulp-imagemin');

gulp.task('optimize_images', ['clean'], function() {
 return gulp.src(['app/images'])
   // Pass in options to the task
   .pipe(imagemin({optimizationLevel: 5}))
   .pipe(gulp.dest(['test/img', 'prod/img']));
});
```

Gulp API: watch

- Last but not least, you can watch files and perform tasks whenever they change.
- For example, as shown below, perform a 'build' task whenever js or css files in your app directory change.

gulpfile.js

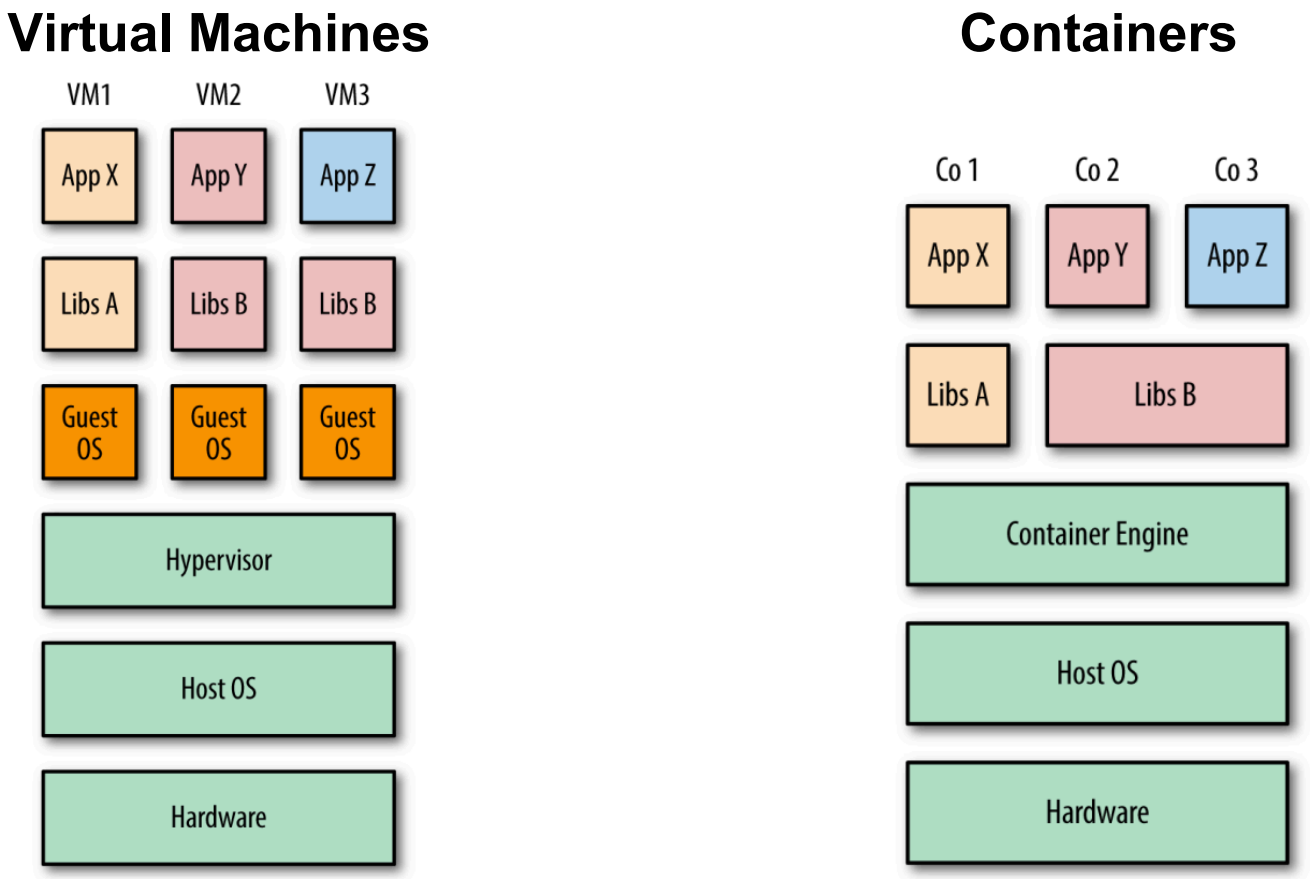
```
const gulp = require('gulp');

gulp.task('my_watch_task', function() {
   gulp.watch(['app/**/*.js', 'app/**/*.css'], ['build']);
});
```

- The watch method is helpful whenever you want to run tasks continuously in the background, allowing you to write code uninterruptedly, and instantly see the changes when you change a file (compiling/transpiling source code, reloading a local web server).

Technische Universität Berlin

USING DOCKER

Containers



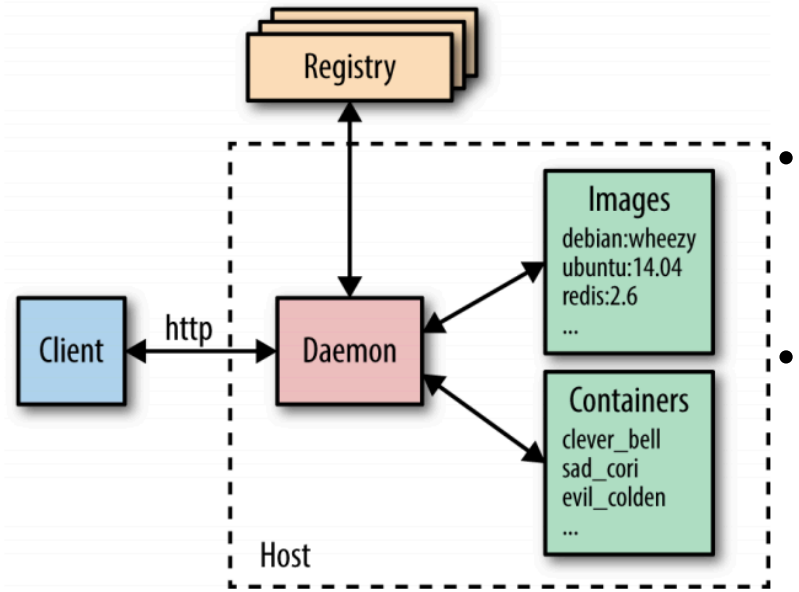
Adrian Mouat: using Docker, O'Reilly, 2016

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Advantages of Containers

Advantage	Benefit
Share resources with host OS	• Increase efficiency, smaller footprint • Run many containers at the same time, emulate production environment
Portability	• Solves “but it worked on my machine” developer issue • Easier transfer of configuration information from providers to (technical) users
Fast startup time	• Caters to developer need of incremental development • Enables new services, like Function-as-a-Service

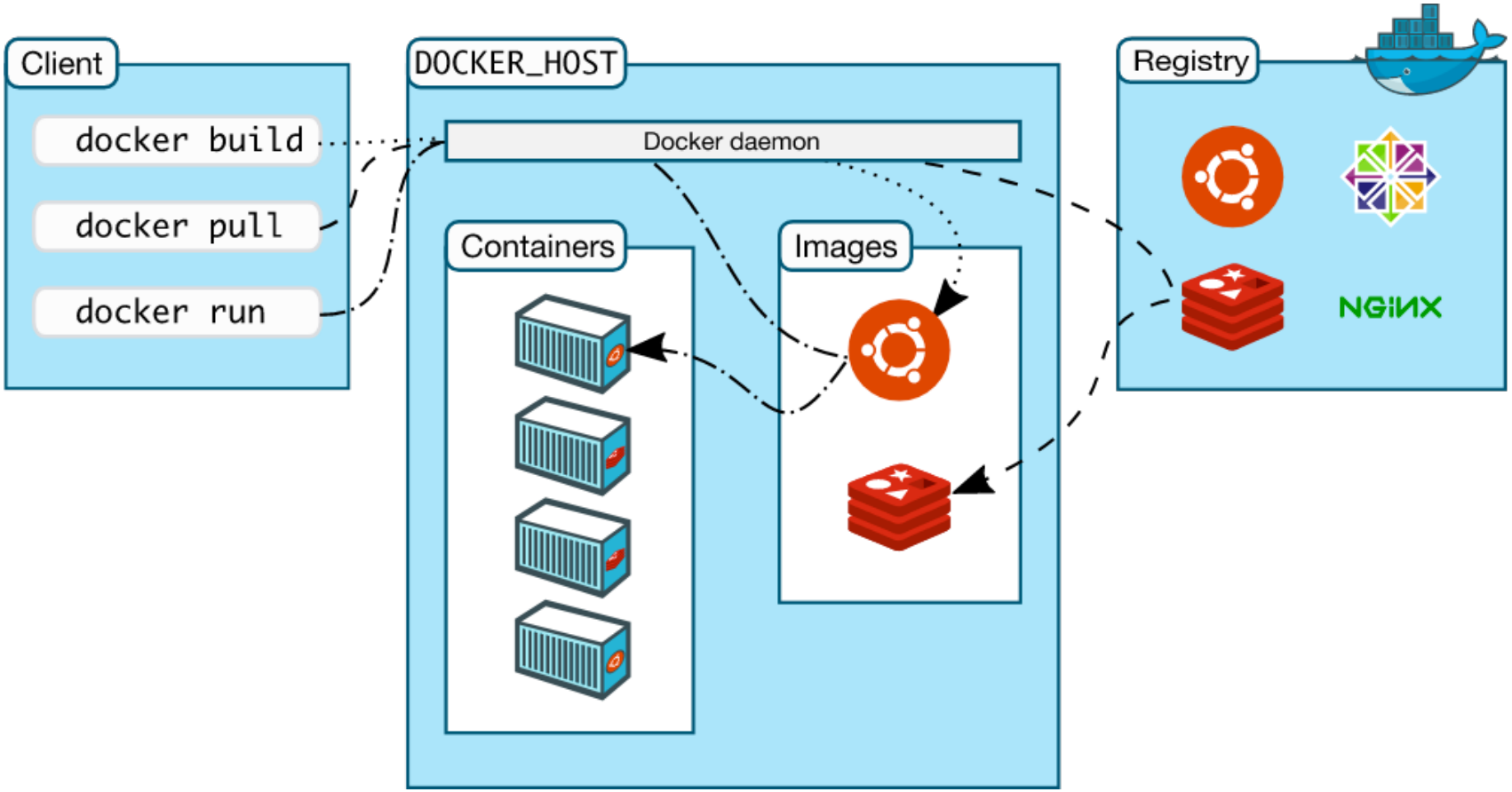
Docker Architecture



- **Docker Daemon**:
  - Create, run, monitor containers
- **Docker Client**
  - Connect to the docker daemon via HTTP
- **Docker Registries**
  - Push (upload) and pull (download) container images
  - By default: Docker Hub Registry

Adrian Mouat: using Docker, O'Reilly, 2016

Docker Architecture



https://docs.docker.com/engine/docker-overview/#docker-architecture

Docker features

- Portable deployment
- Tools for build automation and versioning (like git)
- Component re-use and sharing
- Integration with CI/CD tools

Docker commands: run

```
$ docker
```

```
docker run ubuntu echo "hello world"
```

```
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
2f0243478e1f: Pull complete
d8909ae88469: Pull complete
820f09abed29: Pull complete
01193a8f3d88: Pull complete
Digest:
sha256:8e2324f2288c26e1393b63e680ee7844202391414dbd48497e9a4fd
997cd3cbf
Status: Downloaded newer image for ubuntu:latest
hello world
```

Docker commands: run (again)

```
$ docker
```

```
docker run ubuntu echo "hello world"
```

```
hello world
```

Docker commands: run with input

$ docker
docker run -h foo -it ubuntu /bin/bash root@foo:/# echo "bar"
bar
root@foo:/# exit
exit

- If you want to keep your container running, you can also start an interactive session which will direct your command prompt to the bash executable inside of your Docker container.
- The session ends when you exit the console and the container stops.

Docker commands: ps

$ docker	$ docker
`docker run -i -t ubuntu /bin/bash`	`docker ps`
	`CONTAINER ID IMAGE COMMAND CREATED ... 141761de11c8 ubuntu "/bin/bash " 4 minutes ago`

- Execute `docker ps` to view the current status of your running containers.
- For example, open two command line windows (or tabs) and type the `docker run` command into the left window and `docker ps` into the right window.

Docker commands: rm

$ docker		$ docker	
docker ps -a		docker rm sleepy_montalcini	
CONTAINER ID ... NAMES 141761de11c8 sleepy_montalcini ...		sleepy_montalcini	

- Show all running and stopped containers.
- Then, remove a stopped container.

Docker commands: rm

```
$ docker
docker rm $(docker ps -a -q)
c789f7372509
dd64308b7858
...
```

- Remove all stopped (“exited”) containers with a single command.

Docker images: list images

```
$ docker
```

```
docker images
```

```
REPOSITORY    TAG         IMAGE ID        CREATED         SIZE
ubuntu        latest      f8d79ba03c00    4 days ago      126.4 MB
```

- List the container images that you have stored locally, i.e., the image that you just downloaded (“pulled”) from the Docker Hub Registry.

Docker images: pull (download) an image

```
$ docker
```

```
docker pull ubuntu:16.04
```

```
16.04: Pulling from library/ubuntu
Digest:
sha256:8e2324f2288c26e1393b63e680ee7844202391414dbd48497e9a4fd
997cd3cbf
Status: Downloaded newer image for ubuntu:16.04
```

Docker images: search images

```
$ docker
```

```
docker search ubuntu
```

```
NAME                      DESCRIPTION                                     STARS  OFFICIAL   AUTOMATED
ubuntu                    Ubuntu is a Debian-based Linux operating s...   4468   [OK]
ubuntu-upstart            Upstart is an event-based replacement for ...   65     [OK]
rastasheep/ubuntu-sshd    Dockerized SSH service, built on top of of...   31                [OK]
...
```

Docker images: remove images

- Free space by removing unused images, such as, the “latest” ubuntu image via the “remove image” command, `docker rmi`.

```
$ docker
docker rmi ubuntu:latest
Untagged: ubuntu:latest
```

- In this case, the remove operation only removed (untagged) the “latest” tag, since “latest” and “16.04” currently point at the same container image. If we remove the “16.04” image as well, we see some “Deleted” operation results:

```
$ docker
docker rmi ubuntu:16.04
Untagged: ubuntu:16.04
Untagged:
ubuntu@sha256:8e2324f2288c26e1393b63e680ee7844202391414dbd4...
Deleted:...
```

Docker build: example

- We build an image from a *Dockerfile* for a simple node express app.
- First, we create an *index.js* file with the express dependency. The app listens on port 8080 and replies “Hello world” to GET requests at “/”.

index.js

```
'use strict';
const express = require('express');
// Constants
const PORT = 8080;
// App
const app = express();
app.get('/', function(req, res) {
   res.send('Hello world\n');
});
app.listen(PORT);
console.log('Running on http://localhost:' + PORT);
```

https://nodejs.org/en/docs/guides/nodejs-docker-webapp/

Docker build: example

- The following *package.json* file specifies the start script and our only dependency, the express framework.

package.json

```
{
    "name": "docker_web_app",
    "version": "1.0.0",
    "description": "Node.js on Docker",
    "author": "First Last <first.last@example.com>",
    "main": "index.js",
    "scripts": {
        "start": "node index.js"
    },
    "dependencies": {
        "express": "^4.13.3"
    }
}
```

https://nodejs.org/en/docs/guides/nodejs-docker-webapp/

Docker build: example

Dockerfile

```
FROM node:6.3-slim

# Create app directory
RUN mkdir -p /myapp
WORKDIR /myapp

# Install app dependencies
COPY app/package.json /myapp/
RUN npm install

# Bundle app source
COPY app /myapp

EXPOSE 8080
CMD [ "npm", "start" ]
```

- A Dockerfile consists of multiple instruction statements, each of which create a new read-only layer of the image that we are creating.
- The first line (FROM) specifies the base image which our own image is based on.
- Next, we create the directory /myapp and make it our current working directory.
- Then we COPY the package.json file into the /myapp directory and install the dependencies by executing npm (which is pre-installed via the node:6.3-slim base image) via a RUN instruction.
- Next, the source code (in this case, only the index.js file is copied into the container image)
- Furthermore, we EXPOSE port 8080 which is the port that our node app binds to (see index.js).
- Finally, a command is defined via CMD for starting the node app.

https://nodejs.org/en/docs/guides/nodejs-docker-webapp/

Docker build: example

- We use this Dockerfile to build a new container image via `docker build`:

```
$ docker
```

```
docker build -t markus/node-app:1.0 .
```

```
Sending build context to Docker daemon 4.096 kB
Step 1 : FROM node:6.3-slim
---> b8d17d679859
Step 2 : RUN mkdir -p /myapp
---> Running in 54de336c1e46
---> 82de5f489809
…
Step 8 : CMD npm start
---> Running in ff9dec78878c
---> 84fda048e038
Removing intermediate container ff9dec78878c
Successfully built 84fda048e038
```

https://nodejs.org/en/docs/guides/nodejs-docker-webapp/

Docker build: example

- Besides the Dockerfile, the docker build command needs a build context: files that are located at a file system path or in a remote github repository. These files are sent recursively to the Docker daemon. In the example above, the build context is ".", i.e., the local directory. You can exclude files from the build context by using a .dockerignore file (which works similar to a .gitignore file).
- Each step creates a new read-only layer on top of the previous one, like an onion. Each successful step ends with a new intermediate container, e.g., in step 8, the CMD instruction is added on top of the layer/container ff9dec78878c, thereby creating a new layer/container 84fda048e038. It is generally recommended to keep the number of instructions small, to avoid bloating the image.

https://nodejs.org/en/docs/guides/nodejs-docker-webapp/

Docker build: example

```
$ docker
```

```
docker run --name napp -p 3000:8080 -d markus/node-app:1.0
```

```
10f8d6304633e25e19a1115e7dcd005291c2c8a55da84942d0bca1f073fa8805
```

```
docker logs napp
```

```
npm info it worked if it ends with ok
npm info using npm@3.10.3
npm info using node@v6.3.1
npm info lifecycle docker_web_app@1.0.0~prestart:
docker_web_app@1.0.0
npm info lifecycle docker_web_app@1.0.0~start:
docker_web_app@1.0.0

> docker_web_app@1.0.0 start /usr/src/app
> node index.js

Running on http://localhost:8080
```

https://nodejs.org/en/docs/guides/nodejs-docker-webapp/

Dockerfile instructions

Instruction	Description
`CMD ["exe","param1","param2"]` *(exec form)*	Runs a default command with parameters when a container starts. If the `ENTRYPOINT` instruction is used in a Dockerfile, all entries of `CMD` are interpreted as parameters of the `ENTRYPOINT` command, *e.g.*, `ENTRYPOINT ["exe","param1"]` and `CMD ["param2","param3","param4"]`
`COPY <src> <dest>`	Copy files from the host source to the container image destination. You can specify multiple sources, and also use wildcards for file name matching, such as * and ?.

https://docs.docker.com/engine/reference/builder/

Dockerfile instructions

Instruction	Description
`ENTRYPOINT ["exe","param1","param2"]`	Works similar to `CMD` and should be used if the Docker image is used as an executable.
`ENV <key1>=<val1> <key2>=<val2>`	Set one or more environment variables. Environment variables are useful for customizing your image when you run it, *e.g.,* configurations that you shouldn't hard-code into the image.

https://docs.docker.com/engine/reference/builder/

Dockerfile instructions

Instruction	Description
`EXPOSE <port>`	The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime. `EXPOSE` does not make the ports of the container accessible to the host. To do that, you must use either the `-p` flag to publish a range of ports or the `-P` flag to publish all of the exposed ports. You can expose one port number and publish it externally under another number.

https://docs.docker.com/engine/reference/builder/

Dockerfile instructions

Instruction	Description
`FROM <base_image>`	The first instruction in a `Dockerfile` which sets the base image. This base image is extended and modified by subsequent instructions in the `Dockerfile`. For fast deployments and sharing images with a lower storage footprint, consider using small base images, such as `debian`. `scratch` is a special base image which is completely empty (it does not even have a shell)

https://docs.docker.com/engine/reference/builder/

Dockerfile instructions

Instruction	Description

`RUN <command>`
 	
or
 	
`RUN ["exe","param1","param2"]`
(exec form)
	Execute a command on the container's shell (command line). Alternatively, specify the command in exec form (as an array of executable and parameters) if you strive for portability of your Docker images.
	The `RUN` instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the `Dockerfile`.
	Layering `RUN` instructions and generating commits conforms to the core concepts of Docker where commits are cheap and containers can be created from any point in an image's history, much like source control.

https://docs.docker.com/engine/reference/builder/

Dockerfile instructions

Instruction	Description
`USER daemon`	The `USER` instruction sets the user name or UID to use when running the image and for any `RUN`, `CMD` and `ENTRYPOINT` instructions that follow it in the Dockerfile. For security reasons, it is advisable to always use a special user with limited file access permissions for running your application.

https://docs.docker.com/engine/reference/builder/

Dockerfile instructions

Instruction	Description
`VOLUME ["/dir"]`	The `VOLUME` instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers. With volumes you can share data between host and one or more containers

https://docs.docker.com/engine/reference/builder/

Dockerfile instructions

Instruction	Description
`WORKDIR /path/to/workdir`	The `WORKDIR` instruction sets the working directory for any `RUN`, `CMD`, `ENTRYPOINT`, `COPY` and `ADD` instructions that follow it in the `Dockerfile`. If the `WORKDIR` doesn't exist, it will be created even if it's not used in any subsequent `Dockerfile` instruction.

https://docs.docker.com/engine/reference/builder/

Technische Universität Berlin

BUILDING MODULAR JAVASCRIPT APPLICATIONS

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Module specifications

- JavaScript did not have native support for modules until very recently (introduced in ES2015).
- Therefore, competing module specifications (formats, patterns, styles) have emerged over the years.

AMD

- Asynchronous Module Definition (AMD) is a module specification for front-end JavaScript code.
- With AMD, each module as well as each module's dependencies can be loaded asynchronously.
- AMD modules are defined with the define method, which takes the following 3 parameters:
  - Optional: a module id, e.g., 'modA'
  - Optional: an array of dependencies (module ids) which are required by the module, e.g., ['modB']
  - A factory function that "wraps around" the module implementation, i.e., the function that is executed when the module is instantiated.

AMD example: define (export) module modA

app/lib.js

```
// AMD style module definition
// modA: module id of the module that we define here
// modB: a dependency that is loaded asynchronously

define('modA', ['modB'], function(b) {
    // This is an async callback function
    // which executes as soon as modB
    // has been retrieved…

    // b.doThings() ...

    // Export module modA.
    return /* value to export (object, array, ...)  */;
});
```

AMD example: require (import) module modA

- AMD modules are imported with the `require()` method:
  - The first parameter is an array of dependencies (modules ids)
  - The second parameter is a callback function that uses the dependencies

main.js

```
// AMD style module loading
require(['modA'], function(a) {
   // a.doStuff()
});
```

CommonJS

- CommonJS is a module format which is the de-facto standard in server-side JavaScript as it is used in a vast majority of Node.js packages.
- Different from AMD, the CommonJS specification is not designed for asynchronous module loading. Server-side dependency resolution is typically synchronous because both the module source and destination a co-located (or in close proximity).

CommonJS

Define a CommonJS-style Node.js module

index.js

```
// CommonJS style module
// definition
let doStuff = function() {
   // does stuff
}

module.exports = {
   doStuff: doStuff
}
```

Import a CommonJS-style Node.js module

main.js

```
// CommonJS style module
import
let m = require('./index.js');
m.doStuff();
```

Node.js: packages or modules?

- An npm package is a file or directory that is described by a package.json file.
- Such a package contains source code that you can publish, retrieve, and configure with npm.
- Node.js supports the CommonJS module pattern and npm packages are usually implemented as CommonJS-style modules.
- Since nearly all node packages are modules, the terms are often used interchangeably.

Module specification heterogeneity

- AMD is a common module specification for front-end JavaScript whereas CommonJS is more common when writing Node.js back-end code.
- When we write a full-stack JavaScript application, we have two competing module specifications (and inconsistent specification of require statements).
- If we want to write a module that can be used both in the front-end and back-end, this can cause headaches.

UMD

- How can this problem be solved? Different approaches come to mind:
  - CommonJS support by front-end bundlers (e.g., with Browserify, Webpack, or JSPM).
  - Transform AMD to CommonJS or CommonJS to AMD.
  - Develop a module specification that is compatible to both AMD and CommonJS.
- Universal Module Definition (UMD) uses the latter approach. UMD is a design pattern that can be used by module developers to make their modules compatible to both ADM and CJS.
- UMD accomplishes this by first checking how the module is loaded and then wrapping the module code in the appropriate format.
- There are a few different UMD templates that implement variations of the generic design pattern.

UMD Template example

Technische Universität Berlin

app/lib.js

```
// UMD style module definition
(function(root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD. Register as an anonymous module.
        define(['b'], factory);
    } else if (typeof module === 'object' && module.exports) {
        // Node, CommonJS-like.
        module.exports = factory(require('b'));
    } else {
        root.returnExports = factory(); // Browser globals
(root=window)
    }
}(this, function(b) {
    function doStuff() { // b.doThings() };
    return {
        doStuff: doStuff // Return public functions
    };
}));
```

UMD Template example

- The module in the example has one module dependency (to module “b”), and it is called as an anonymous function expression after performing an if-else check:
  - If the code that uses the module has a `define` function, then the module is defined using the AMD module style.
  - If, on the other side, a module object exists, then the module is defined using the CommonJS module style.
  - Else, the example above returns a global module named `returnExports`.

Module loaders

- Module loaders are libraries for resolving module dependencies and loading JavaScript modules in a web browser.
- Remember that web browsers do not natively support a single module specification that we discussed before: neither AMD, nor CommonJS, nor ES2015 modules.
- With a module loader library that you need to load and initialize first, you can then load modules in a web browser.
- There are two main module loaders out there:
  - Require.js was the first module loader for JavaScript and has been used successfully in many projects.
  - System.js is gaining traction because it addresses some limitations of Require.js, such as a more comprehensive support for different module specifications.

RequireJS

- RequireJS is an AMD module loader library which pioneered modular JavaScript front-end development.
- Let’s say we have 2 JavaScript files: lib.js and main.js.
  - In lib.js we define an AMD module with module id 'modA' and no dependencies
  - In main.js we load the module.

RequireJS example

Technische Universität Berlin

app/lib.js

```
// AMD style module definition of a module without
// dependencies []
define('modA', [], function() {
    var ratingModule = {
        rating: function(stars) {
            if (stars > 3) {
                return "great";
            } else {
                return "nice";
            }
        }
    };

    return ratingModule;
});
```

RequireJS example (2)

Technische Universität Berlin

app/main.js

```
requirejs.config({
    baseUrl: './',
    paths: {
        // the left side is the module ID,
        // the right side is the path to
        // the module files, relative to baseUrl.
        // Also, the path should NOT include
        // the '.js' file extension.
        modA: 'app/lib',
        jquery: 'node_modules/jquery/dist/jquery.min'
    }
});

require(['modA', 'jquery'], function(a, $) {
    $("#test").html("<h2>AMD + Require.js is " + a.rating(4) +
".</h2>");
});
```

RequireJS example (3)

- Include a script tag in your HTML page that sources the require.js library. Add a data-main attribute to the script tag that references the entry point, i.e., the first module which then might lazy-load additional modules.

index.html

```
<html>
<head>
   <meta charset="UTF-8">
   <title>Module exercise</title>
</head>
<body>
   <div id="test">Loading...</div>
   <script data-main="app/main.js"
       src="node_modules/requirejs/require.js">
   </script>
</body>
</html>
```

Module bundlers

- Modularization helps organizing and maintaining your code base. However, what is good in development can be bad in production.
- A highly modularized JavaScript application that synchronously loads one module at a time from a remote server could potentially result in bad client-side performance and a bad user experience.
- Before you deploy your code to production, it should be reorganized to improve client-side load times.
- This is where module bundlers come into play, such as:
  - Browserify, or
  - Webpack

Browserify

- Released in 2011, Browserify was the first JavaScript module bundler with support for CommonJS modules.
- A main advantage of CommonJS support is that you can use (require) Node.js packages in your front-end code.

Browserify example

- Given a CommonJS-style module

app/lib.js

```
// CommonJS style module definition
var rating = function(stars) {
    if (stars > 3) {
        return "great";
    } else {
        return "nice";
    }
}

module.exports = {
    rating: rating
}
```

Browserify example

- … and a file that imports the CommonJS-style module:

app/main.js

```
var lib = require('./lib.js');
var $ = require('jquery');


$("#test").html("<h2>CJS + Browserify is " + lib.rating(4) +
".</h2>");
```

Browserify example

- Run the command line:

```
browserify ./app/main.js -o ./bundle.js
```

- ... and load the JavaScript payload in your html page:

index.html

```
<html>
<head>
   <meta charset="UTF-8">
   <title>Module exercise</title>
</head>
<body>
   <div id="test">Loading...</div>
   <script src="bundle.js"></script>
</body>
</html>
```

Webpack

- Webpack works similarly, however, provides more configuration options: simply execute `$ webpack`
- A simple (equivalent) Webpack configuration in our example:

webpack.config.js

```
// webpack.config.js
module.exports = {
    entry: './app/main.js',
    output: {
        path: './',
        filename: 'bundle.js',
    },
    resolve: {
        modulesDirectories: ['node_modules'],
    },
};
```

Technische Universität Berlin

JAVASCRIPT AND THE NETFLIX USER INTERFACE

http://queue.acm.org/detail.cfm?id=2677720

A/B Testing Netflix.com

- Netflix culture of A/B testing
  - Targets of A/B testing: movie personalization algorithms, video encoding, User Interface, etc. (all elements of the Netflix service)
  - It is not unusual to find the typical Netflix subscriber allocated into 30 to 50 different A/B tests simultaneously.
  - The tests allow experimentation with radically different UI experiences from subscriber to subscriber, and the active lifetime of these tests can range from one day to six months or more.
  - The goal is to understand how the fundamental differences in the core philosophy behind each of these designs can enable Netflix to deliver a better user experience.

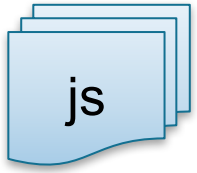
A/B Testing Netflix.com

- A/B testing on the Netflix Web site tends to add new features or alter existing features to enhance the control experience.
  - Many of the Web-site tests are designed to be “stackable” with other A/B tests.
  - This ensures that newly introduced functionality can coexist with other tests.
  - It should be noted that the testing encompasses all pieces of the Netflix UI (HTML, CSS, and JavaScript), but the focus here is on using JavaScript to reduce the scope of the problem.

Facets to Features to Modules

Technische Universität Berlin

Average Website



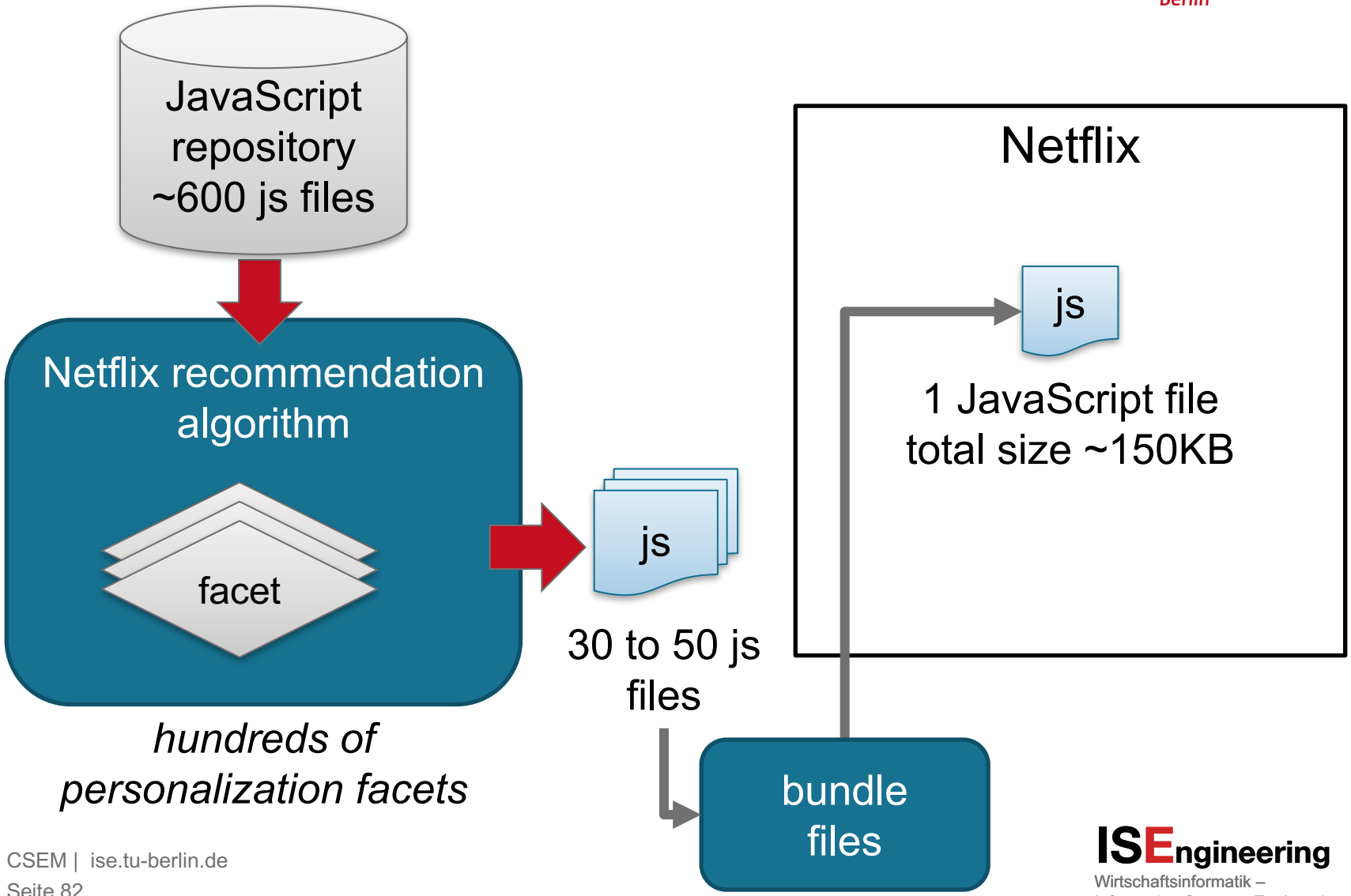
18 JavaScript files
total size ~290KB

Netflix

1 JavaScript file
total size ~150KB

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Facets to Features to Modules



Facets to Features to Modules

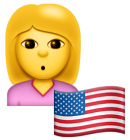
- These facets can be commonly derived via a subscriber's
  - A/B test allocations,
  - country of signup,
  - viewing tastes, and
  - sharing preferences (Facebook integration),
  - or by any piece of arbitrary logic.
- These facets act as switches, a method by which the UI can be efficiently pivoted and tweaked.

Challenge:

→ **how to manage packaging and delivery of many different UIs in a maintainable and performant manner**.

Facets to Features to Modules: A simple example

Experience	Desired Behavior	Requirements
Search Box Test Cell 0 (Control)	Search Results Page	Must be a U.S. subscriber
Search Box Test Cell 0 (Control)	Search Results Page Type 2	Not a U.S. subscriber
Search Box Test Cell 1	Autocomplete	Subscriber allocated to Cell 1
Search Box Test Cell 2	Instant Search	Subscriber allocated to Cell 2




search …

Terminator 1

Terminator 2

Terminator 3

search …

Terminator 1

Terminator 2

Terminator 3

Terminator

Terminator 1

Terminator 2

Terminator 3

Terminator

Terminator 1

Terminator 2

Terminator 3

Facets to Features to Modules

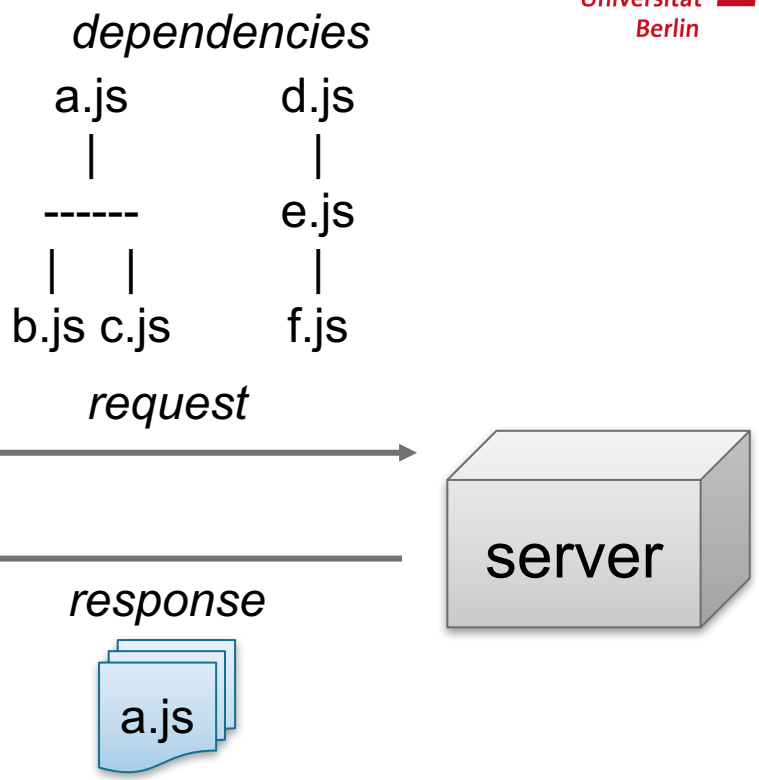
- These are three distinct experiences, or "features," with each one being gated by a set of very specific personalization facets.
  - Other parts of the page, such as the header or footer, can be tested in a similar manner without affecting the search-box test.
  - Under this test stratagem, it is imperative to separate each functional section of the Web site into discrete sandboxed files (= JavaScript modules).
  - Modules also allow seamless feature portability from one page to the next. A Web page should be divided into smaller and smaller pieces until it is possible to compose new payloads using existing modules (single responsibility principle).
  - Using modules to encapsulate features provides the ability to build an abstraction layer on top of the personalization facets that gate the A/B tests.

Dependency Management

- Modules also allow dependency management.
  - In many languages, dependencies can be imported synchronously, as the runtime environment is colocated on the same machine as the requested dependencies.
  - The complexity in managing browser-side JavaScript dependencies, however, is that the runtime environment (browser) is separated from its source (server) by an indeterminate amount of latency.
  - Network latency is arguably the most significant bottleneck in Web application performance today, so the challenge is in finding the balance between bandwidth and latency for a given set of indeterministic constraints that may differ per subscriber, per request.

Dependency Management: Synchronously loading all dependencies

```
1   <html>
2
3   <head>
4       <meta charset="UTF-8">
5       <title>Simple approach</title>
6   </head>
7
8   <body>
9       <script src="a.js"></script>
10      <script src="b.js"></script>
11      <script src="c.js"></script>
12      <script src="d.js"></script>
13      <script src="e.js"></script>
14      <script src="f.js"></script>
15  </body>
16
17  </html>
```

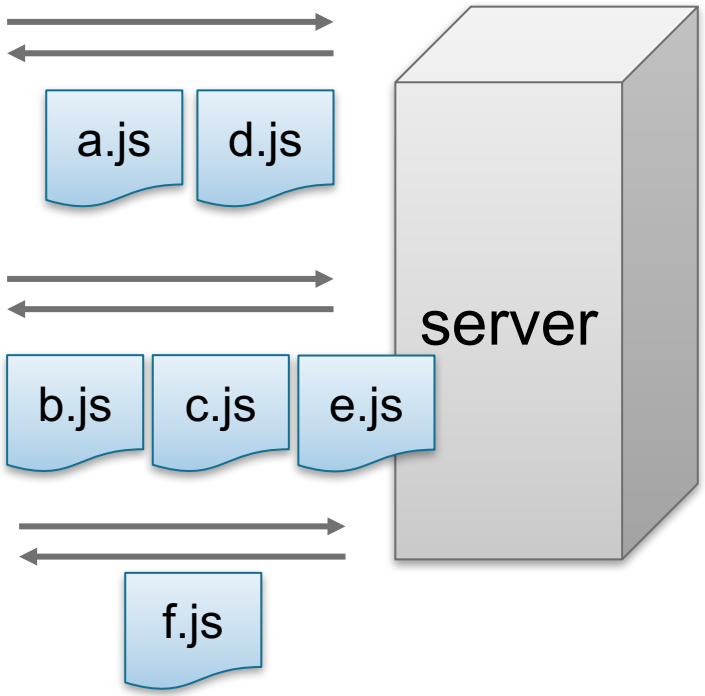


- Approach: include all dependencies on the page, regardless of whether or not the module is used

⛔ penalizes users across the board, with bandwidth constraints often exacerbating already long load times

Dependency Management: Asynchronously loading dependencies

```
1   <html>
2
3   <head>
4       <meta charset="UTF-8">
5       <title>Simple approach</title>
6   </head>
7
8   <body>
9       <script src="a.js"></script>
10      <script src="d.js"></script>
11  </body>
12
13  </html>
```



- Approach: making multiple asynchronous requests back to the server as it determined missing
- ⛔ penalizes deep dependency trees: for a tree of depth N, up to N-1 Round Trip Times (RTTs) are necessary

Dependency Management with AMD + RequireJS

- Use of AMD libraries such as RequireJS allows users to create modules, then preemptively generate payloads on a per-page basis by statically analyzing the dependency tree.
  - This solution combined the best of both previous solutions by generating specific payloads containing only the things needed by the page and by avoiding unnecessary penalization based on the depth of the dependency tree.
  - More interestingly, users can also opt out entirely from the static-analysis step and fall back on asynchronous retrieval of dependencies, or they can employ a combination of both.

Dependency Management with AMD + RequireJS

```
define('depA', function() {
        // execute some code
});
define('depB', function() {
        // execute some code
});
define('foo', ['depA', 'depB', 'depC'], function(a, b, c) {
        // this function is an async callback.
        // code here only runs after A, B, C have loaded.
        // A and B are loaded as part of page payload,
        // but C will be fetched asynchronously since it has not yet
        // been defined by the time we reach module foo.
        // thus, code here is NOT run until C has been retrieved
        // and executed.

        // the value of this return statement is what this module exports.
        // it is the value imported by other modules when this module is
        // requested as a dependency
        return …;
});
```

```
foo
|__depA
|__depB
|__depC (fetched asynchronously)
```

Search Box Example with AMD + RequireJS

- In our search box A/B test example there are 3 distinct search box features (user experiences)
- How do you load only the correct search box experience for a given user (in either cell 0, cell 1, or cell 2)?
  - Synchronously loading all dependencies?
  - Asynchronously loading dependencies (on demand)?

Search Box Example with AMD + RequireJS: Synchronously loading all dependencies

- It is possible to add all the boxes to the payload, then have the parent module add logic that allows it to determine the correct course of action.

```
define('header', ['controlSearch', 'autocompleteSearch', 'instantSearch', 'facets'],
    function(control, autocomplete, instant, facets) {
        // facets is a dependency that contains personalization facets
        if (facets.inSearchTest && facets.geo !== 'US') {
            autocompleteSearch.init();
        } else if (facets.inSearchTest && facets.geo === 'US') {
            instantSearch.init();
        } else {
            controlSearch.init();
        }
    }
);
```

- This is unscalable: loading all possible dependencies increases the payload size, thereby increasing the time it takes for a page to load.

Search Box Example with AMD + RequireJS: Asynchronously loading dependencies

- A second option of fetching dependencies just in time is possible but may introduce arbitrary delays in the responsiveness of the UI.
- In this option, only the modules that are needed are loaded, at the expense of an additional asynchronous request.
- If any of the search modules has additional dependencies, there will be yet another request, and so on, before search can be initialized.

Search Box Example with AMD + RequireJS: Asynchronously loading dependencies

Technische Universität Berlin

```
define('header', ['facets'], function(facets) {

        // facets is a dependency that contains personalization facets
        if (facets.inSearchTest && facets.geo !== 'US') {
                // fetch the autocomplete dependency over the wire
                // search cannot be used until this file and all of the
                // autocomplete search dependencies are loaded.
                require(['autocompleteSearch'], function(autocompleteSearch) {
                        // async callback
                        autocompleteSearch.init();
                });
        } else if (facets.inSearchTest && facets.geo === 'US') {
                // fetch the instant search dependency over the wire
```

etc..

Search Box Example with AMD + RequireJS

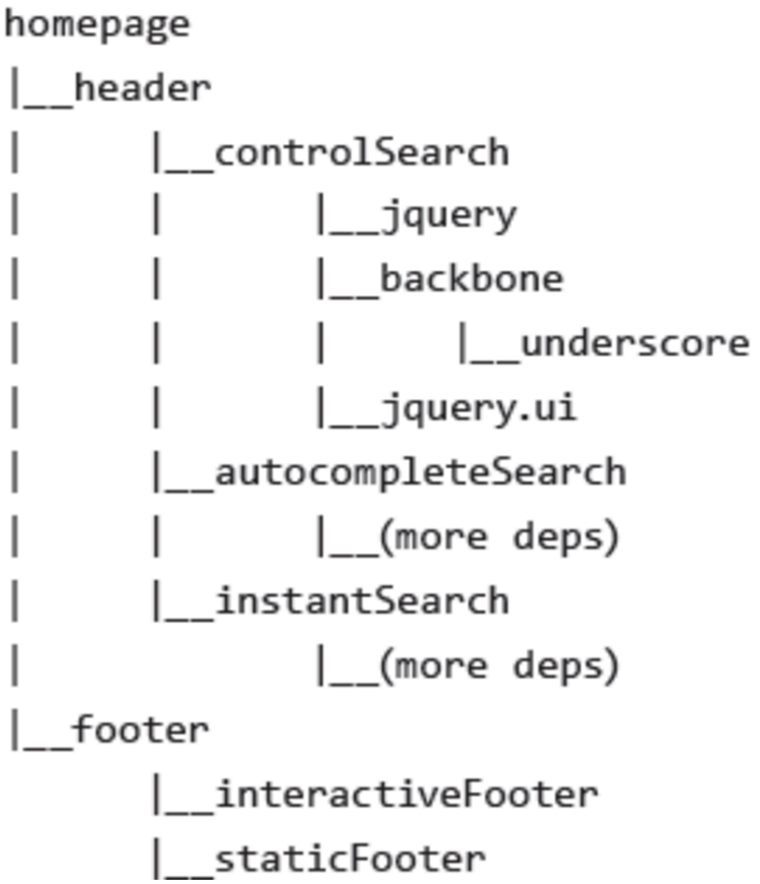
- Both options are undesirable and have proven to have a significant negative impact on the user experience.
- They also do not take into account the possibility that certain personalization facets are available only on the server and for security reasons cannot be exposed to the JavaScript layer.

A/B testing @ Netflix scale

- The Web site deploys a new build on a weekly cycle.
- For every build cycle, the Web site generates approximately 2.5 million unique combinations of JavaScript and CSS payloads.
- Given this huge number, it is tempting to go the route of letting the browser fetch dependencies as the tree is resolved.
  - This solution works for small code repositories, as the additional serial requests may be relatively insignificant.
  - However, a typical payload on the Web site contains 30 to 50 different modules because of the scale of A/B testing. Even if the browser's parallel resource fetching could be leveraged for maximum efficiency, the latency accumulated across a potential 30-plus requests is significant enough to create a suboptimal experience.

A/B testing @ Netflix scale

- Even with a significantly simplified example with a depth of only five nodes (figure on the right), the page will make four asynchronous requests before the page is ready.
- A real production page may easily have 15-plus depth.



A/B testing @ Netflix scale

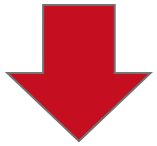
- Since asynchronous loading of dependencies has already been disqualified for this particular situation, it becomes clear that the scale of A/B testing dictates the choice to deliver a single JavaScript payload.
- If single payloads are the solution, this might give the impression that these 2.5 million unique payloads are generated ahead of time.
  - This would necessitate an analysis of all personalization facets on each deployment cycle in order to build the correct payload for every possible combination of tests.
  - If subscriber and A/B testing growth continues on its correct trajectory, however, then preemptive generation of the payloads will become untenable.
  - The number of unique payloads may be 2.5 million today, but 5 million tomorrow. It is simply not the correct long-term solution for Netflix's needs.

Conditional Just-in Time Dependency Resolution

- What the A/B testing system needs, then, is a method by which conditional dependencies can be resolved without negatively affecting the user experience.
- In this situation, a server-side component must intervene to keep the client-side JavaScript from buckling under its own complexity.
- Since we are able to determine all possible dependencies via static analysis, as well as the conditions that trigger the inclusion of each dependency, the best solution given our requirements is to resolve all conditional dependencies when the payload is generated just in time.

Conditional Just-in Time Dependency Resolution in the Search Box Example

Experience	Desired Behavior	Requirements
Search Box Test Cell 0 (Control)	Search Results Page	Must be a U.S. subscriber
Search Box Test Cell 0 (Control)	Search Results Page Type 2	Not a U.S. subscriber
Search Box Test Cell 1	Autocomplete	Subscriber allocated to Cell 1
Search Box Test Cell 2	Instant Search	Subscriber allocated to Cell 2



JavaScript modules

Experience	Desired Behavior	Requirements	Module Name
Search Box Test Cell 0 (Control)	Search Results Page	None	controlSearch
Search Box Test Cell 1	Autocomplete	Not a US subscriber	autocompleteSearch
Search Box Test Cell 2	Instant Search	Must be a US subscriber	instantSearch

Conditional Just-in Time Dependency Resolution in the Search Box Example

- The browser has asked for the homepage JavaScript payload.
- There is a dependency tree created as a result of static analysis.

```
define('search', function() {
        // search has no dependencies.
        // which search module goes here?
});
define('header', ['search'], function(search) {
        search.init();
});
define('footer', function() {
        // footer has no dependencies.
        ...
});
define('homepage', ['header', 'footer'], function(header, footer) {
        ...
});
```

```
homepage
|__header
|       |__search
|__footer
```

Conditional Just-in Time Dependency Resolution in the Search Box Example

- There is a table that maps the search module to three potential implementations.
- Since the header cares only about the inclusion of a search module, but not its implementation, we can plug in the correct search module by ensuring that all implementations conform to a specific contract (i.e., a public API).

```
// controlSearch.js
define('search', function() {
        // the control search implementation here
        return { init: function(){...}};
});

// autocompleteSearch.js
define('search', function() {
        // the autocomplete search implementation here
        return { init: function(){...}};
});

// instantSearch.js
define('search', function() {
        // the instantSearch search implementation here
        return { init: function(){...}};
});
```

Conditional Just-in Time Dependency Resolution in the Search Box Example

- Having variations of a single experience conform to a similar public API allows us to change the underlying implementation by simply including the correct search module.
- Unfortunately, because of JavaScript's weakly typed nature, there is no way to enforce this contract, or even to verify the validity of any modules claiming to conform to said contract.
- The responsibility to do the right thing is often left up to the developers creating and consuming these shared modules. In practice, nonconforming modules are not game breakers; "drop-in" replacements as in the previous example are typically entirely self-contained with the exception of a single entry point, which in this case is the exposed `init()` method.
- Modules with complex public APIs tend to be shared as common libraries, which are less likely to be A/B tested in this manner.

Conditional Just-in Time Dependency Resolution in the Search Box Example

- When a request comes in asking for the homepage payload, we already know all the possible files the subscriber may receive, as a result of static analysis.

```
homepage
|__header
|       |__controlSearch
|       |__autocompleteSearch
|       |__instantSearch
|__footer
```

Conditional Just-in Time Dependency Resolution in the Search Box Example

- As we begin appending files to the payload, we can look up in the search-box test table whether or not this file is backed by an eligibility requirement (i.e., whether the subscriber is eligible for that feature).
- This resolution will return a Boolean value, which is used to determine if the file gets appended.

Experience	Desired Behavior	Requirements	Module Name
Search Box Test Cell 0 (Control)	Search Results Page	None	controlSearch
Search Box Test Cell 1	Autocomplete	Not a US subscriber	autocompleteSearch
Search Box Test Cell 2	Instant Search	Must be a US subscriber	instantSearch

true / false

```
homepage
|__header
|       |__~~controlSearch~~
|       |__autocompleteSearch
|       |__~~instantSearch~~
|__footer
```

Conditional Just-in Time Dependency Resolution at Netflix

- Using a combination of static analysis to build a dependency tree, which is then consumed at request time to resolve conditional dependencies, we're able to build customized payloads for the millions of unique experiences across Netflix.com.
- It's important to note this is only the first step in a chain of services that finally delivers the JavaScript to the end user.